GEOMETRIC ABSTRACTIONS FOR CONCEPTUAL DESIGN SUPPORT

by

GEORGIOS A. MARGELIS

B.S., Hellenic Naval Academy (1986)

Submitted to the Departments of Ocean Engineering and Civil and Environmental Engineering
in partial fulfillment of the requirements for the degrees of

Ocean Engineer

and

Master of Science in Civil and Environmental Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1994

Author .....
Departments of Ocean Engineering and Civil and Environmental Engineering, May 12, 1994

Certified by .....
Duvvuru Sriram, Associate Professor, Thesis Supervisor

Certified by .....
Nicholas M. Patrikalakis, Associate Professor, Thesis Reader

Accepted by .....
A. Douglas Carmichael, Chairman, Ocean Engineering, Departmental Committee on Graduate Students

Accepted by .....
Joseph M. Sussman, Chairman, Civil and Environmental Engineering, Departmental Committee on Graduate Students

GEOMETRIC ABSTRACTIONS FOR CONCEPTUAL DESIGN SUPPORT

by

GEORGIOS A. MARGELIS

Submitted to the Departments of Ocean Engineering and Civil and Environmental Engineering
on May 12, 1994, in partial fulfillment of the
requirements for the degrees of
Ocean Engineer
and
Master of Science in Civil and Environmental Engineering

Abstract

In order to support the conceptual design of engineering objects, the computer module GAB (Geometric ABstractions) was developed in C++ language, on top of the non-two manifold solid modeler GNOMES. The module provides the capability of creating a set of engineering objects, such as beams, plates, shells, etc., widely used in Ocean and Civil engineering. In addition it allows evolving shape design, through different levels of abstraction. This means that the user can display the objects from the initial stages of design, before all the information is known. This can be achieved through the different levels of abstraction, which have been implemented as C++ classes in a hierarchical manner. The hierarchical tree of the classes has been designed based on a standard shape classification. At the top level there is the base class which contains all the utilities for the classes. Further it allows the user to assign to each object features like bounding box, type of material, color, specific weight. At the next level there are the classes for the straight line, curved line, surface, and curved surface and also all the primitive objects like cuboid, cylinder, etc. From the straight line all the beams, pipes and long objects are derived by applying to it different cross sections. From the surface all the plates and surface-like objects are derived, while from the curved surface, all the three dimensional curved objects and shells are derived.

The GAB module provides the capability of creating user defined objects, through their parametrization. Also an interface class was designed that serves as a reference to the geometry, contains domain knowledge about default values for attributes, and interaction interfaces with various client applications. A user interface called GRAPHITI, was also developed and used in this work.

Thesis Supervisor: Duvvuru Sriram,
Title: Associate Professor
Thesis Reader: Nicholas M. Patrikalakis,
Title: Associate Professor

Acknowledgments

I would like to express my gratitude to the Hellenic Navy for giving me the opportunity to study at M.I.T. for three years and supporting me financially during these years.

I would like to thank Professor Duvvuru Sriram for his continuing support and guidance during this thesis. He was always present to help me out of the many difficulties I encountered. His guidance and experience were invaluable in leading me through the area of Computer Aided Design, where I did my thesis.

I would like to thank Professor Nicholas Patrikalakis for his strong support and valuable advice. Particularly I would like to thank him for accepting to become the reader of my thesis. His experience and guidance helped me to plan my graduate education.

I would like to thank Gorti Sreenivasa Rao, for his every day support and guidance that led to the completion of this thesis. He was always present to help me along the way, and without his extraordinary encouragement, suggestions and efforts I would have never been able to complete this work. I am most grateful to him for all his help throughout this work.

I would like to thank CAPT Alan Brown USN for his continuing support and guidance as my 13A program advisor for two years. His guidance and experience were invaluable in leading me through my academic education. Also I would like to thank LCDR Jeffrey Reed USN for his valuable advice and for being my 13A program advisor for the last year of my studies.

Special thanks to Spyridon Maragos, George Margetis, Athanassios Tjavaras for answering all the questions I had and giving me valuable advice.

Special thanks to the 13A students of my class: CDR Roderick Lusted USN, CDR John Orzalli USN, LCDR Francis Colberg USN, LCDR Melissa Smoot USN, LCDR Grecory Thomas USN, LT Mark Bracco USN, LT Stephen Markle USN, LT David Fox USN for helping me during the last three years with all the problems I encountered.

Contents

List of Figures

Chapter 1

Introduction

1.1 Chapter Overview

The design of new engineering objects is a complicated process. All the information is not known from the initial stages, but it is acquired during the design process. Also the designers want to associate some properties or features with each object other than geometric representations. Therefore a system that would support conceptual design from the initial stages and would allow the inclusion of new information during the design, through different levels of abstraction, would be very useful. Further, each engineering discipline has different objects associated with it, therefore it would be very useful to create different computer modules for each discipline, with the objects most widely used in this particular discipline.

Traditional CAD systems require a complete knowledge of the geometry being represented, while in this thesis we discuss the notion of abstract and incomplete geometry representation. In order to support the conceptual design in the areas of Ocean and Civil engineering, we built the GAB (Geometric ABstractions) computer module. GAB is built on top of a non-two manifold solid modeling engine, and designed to support conceptual design. It allows the creation of objects commonly used in Ocean and Civil Engineering and their representation in different levels of detail, based on the available information. Also it allows the parametrization of objects for the creation of user defined shapes. The GAB module is written in the

C++ computer language.

1.2 Motivation

This thesis addresses the issue of knowledge representation for geometry. Traditional CAD systems, store this knowledge in the form of primitive building blocks, like cuboids, cylinders, spheres, etc for a CSG-based solid modeler. These primitive blocks can lead to the synthesis of a great variety of objects [16]. Nevertheless they do not relate directly to any engineering domain but have a general form for any domain. This thesis examines the knowledge representation in the domains of Ocean and Civil engineering. Also traditional CAD systems require a complete knowledge of the geometry being represented, while this thesis discusses the notion of abstract and incomplete geometry representation, that would support the conceptual design stage.

Engineering design is an iterative and a collaborative process. The information for various designed objects is generated through various and several stages of this process. Initially, only a general idea of the new product exists. Then, usually through elaborate calculations, more knowledge is acquired about the exact shape. So designers in engineering need a computer program that will help them visualize the objects through the various design steps. However, each area of engineering uses numerous and different kinds of objects. So a generic system would not be able to provide support for all the engineering disciplines. Therefore the creation of a separate computer module for each area of engineering would be more useful. In this thesis, we discuss the role of knowledge representation to provide domain-specific information, that could be used by a knowledge-based design agent. For example the provision of different type of beams or plates in the computer module, would be very helpful for naval architects that design ships and it would help them save a lot of time and would also provide a useful tool for visualizing the objects created.

Since information is acquired slowly through elaborate steps, the representation of objects on the screen should also provide the capability for the slow inclusion of the information on the object displayed and the redisplay of this object each time

new information is added. If the engineer has to acquire all the information and at the end of the design is able to display the object, then this is not very useful. What designers need is a system that would help them during the design process and not after they have finished. We discuss partial representation of objects to support conceptual design. We also present a framework which allows an intelligent design agent to communicate with this geometry.

1.3 Objectives

In the previous section the problem was described in detail. The solution of this problem would require the creation of a computer module that should have the following properties:

- it should support conceptual design from the initial stages to the very end, through various levels of abstraction. It should also allow evolving descriptions of objects during the design process.
- the module should provide engineering objects for each particular area of engineering separately. Since the area of engineering we are interested in is Ocean and Civil Engineering, then the computer module should provide for the engineering objects mainly used in these two areas. The user should be able to rapidly construct these engineering objects directly.
- the module should retain the flexibility to allow the parametrization of arbitrary user defined shapes.
- the system should allow persistent storage and retrieval of design information.
- the module should be fully supported on a CAD modeler.

1.4 Roadmap of the Thesis

This chapter has introduced the primary objective of this study which is to create a computer module that would allow ocean and civil engineers to create engineering

objects through different levels of abstraction in order to allow gradual shape evolution. Also this chapter contains a detailed description of the problem addressed by this thesis, and why this problem is important.

The second chapter provides a brief description of the background required. We briefly describe the concepts of solid modeling, object oriented programming, features and form features, and the engineering design process. Also this chapter includes a brief description of the GNOMES non-manifold geometric modeler.

The third chapter provides a description of the concept of levels of abstraction in general, and also it gives a generic description of the Geometric Abstraction Module (GAB). It provides information about the object classification method that was used in GAB to support the design through the different levels of abstraction.

The fourth chapter provides a detailed description of GAB classes. For each class a general description is provided, together with the description of all the attributes and methods of this class. Also this chapter explains the concept of a parametrized object and it describes the class abstraction that provides this facility. We also describe the concept of a wrapper class, which acts as an interface to the client applications. Finally there are some examples from the creation of engineering objects.

The fifth chapter provides a description of the GRAPHITI user interface for the GNOMES solid modeler. This interface was initially built using the Phigs graphics library, but eventually it was changed to the Hoops graphics library.

Finally, the sixth chapter summarizes our conclusions and identifies future work.

At the end of the thesis there are four appendices. The first one contains the GNOMES hierarchical construction. The second one contains the header files for all the GAB classes. The third one includes drawings with the dimensions of the engineering object shapes, especially beams and engineering assemblies, that are used in the classes of the GAB module. The last one contains various screen dumps, depicting GAB object generation.

1.5 Chapter Summary

Engineers design new objects, through a complicated process. Initially they do not know all the information, but they acquire it during the design. Also they use different objects in each engineering discipline. Therefore designers need systems that would provide for the gradual inclusion of information and the evolving shape description and also would include objects used in the different engineering disciplines.

The objective of this thesis is the creation of a system that would support the evolving shape description through various levels of abstraction, in the domains of Ocean and Civil Engineering.

Chapter 2

Background

2.1 Chapter Overview

In this chapter, we provide some discussion of the various technical areas that we draw upon in this thesis. In section 2.2, we discuss the object oriented programming paradigm. It is a different philosophy of programming, which uses the concept of objects and messages. Objects have attributes and methods, which represent the properties and behavior of the object. The object oriented approach allows reusability, flexibility and easy expansion.

Solid modeling is a technique used to represent objects, that contain not only geometric information, but also topological information about the connectivity of the shapes that constitute the object. Section 2.3 discusses three different kinds of solid models, decomposition model, constructive model, and boundary model.

In section 2.4, we present the Selective Geometric Complexes (SGC) model ([21]), a non-manifold model based on cells, where cells are connected open subsets of an extent. The cells can be set to active or non active status and this allows the creation of point sets which may be open or unbounded or non homogeneous.

GNOMES is a non two manifold geometric modeler based on the Selective Geometric Complexes (SGC) paradigm. It is a combination of CSG and Boundary representation modelers. GNOMES is written in the C++ language and it uses the EXODUS database management system [20],[1].

In Section 2.6, we describe the concept of features. Features can be considered as higher level modeling elements, compared to the low level primitive geometric elements that were used until now as the building blocks of objects.

The engineering design process involves three different stages: the functional design, the conceptual design and the detailed design. At the functional design there is no notion of geometry, while at the conceptual design the engineer selects components and subassemblies and defines their relationship. The detailed design involves the calculation of the dimensions and properties of each component and subassembly.

Literature survey was performed in order to find out the concepts that are used in the existing systems. Different kind of systems were found that fulfill some of the objectives mentioned in Section 1.3 of the first chapter. But no module was found that allows shape evolution through different levels of abstraction, that would give the user the flexibility to display the objects during the design process and support the evolutionary design process. GAB was designed to address some of these requirements.

2.2 Object Oriented Programming

The object oriented paradigm is a philosophy of programming which involves the use of objects and messages. Objects are entities that combine the properties of procedures and data, since they perform computation and save local state [27].

Every object has a unique identity and also attributes and methods attached to it. The attributes represent the properties of the object, while the methods represent its behavior. The methods can be called by passing messages to the object that the method is attached to. Objects are grouped into classes with similar properties and behavior. The object oriented programming paradigm has many advantages, but the major ones are that it allows reusability, it is very flexible to change in problem specifications, it allows easy expansion and it can easily model the problem concepts.

In object oriented programming, objects with similar behavior and properties are grouped into one class. So a class is a template of objects, and an object is said to

be an instance of a class. Classes are defined hierarchically, in a manner that allows different kind of links between them, which can be a parent - child relational link, or a friend - friend relation, or no relation at all. An object is said to be an instance of a class if it has the behavior and properties that are described by this class. Classes are defined hierarchically, in a manner that allows different kind of links between them.

The object oriented paradigm also incorporates the concept of data abstraction by structuring the knowledge in different classes through various levels. Each of these levels in the class hierarchy is a different level of abstraction of the knowledge. So the object oriented paradigm fits well with the purpose of this study which is the creation of engineering objects through different levels of abstraction.

Reusability and extensibility are some of the major advantages of object oriented programming, because the same classes can be reused many times or even extended by the creation of new subclasses. Also the object oriented approach leads to an implementation which is modular.

Modeling using an object oriented approach requires the following stages ([22]):

- Analysis:

  At this stage an analysis model is created. This model takes into account real-world concepts and does not involve programming issues, but it models the application domain's properties and behavior. This is the stage where all the knowledge is acquired from the real world and thus this model is an abstraction of the real world system. Also it involves the acquisition of the requirements for the functionality that the object oriented system should have.

- System design:

  This stage involves the development of the overall architecture of the system and the subsystems. This stage can be incorporated in the object design, especially for small models.

- Object design:

  Object design involves the incorporation of details into the model created in the

analysis stage. All the programming issues are considered in this stage. Also all the algorithms and data structures are designed.

- Implementation:

  The implementation of the object design into an object oriented computer language is performed in this stage, which should be the easiest one, because it involves the translation of the object design to computer code.

2.3 Solid Modeling

Geometric modeling is a technique used to represent geometric shapes in the computer. There are three different kinds of representation: wireframe, surface, and solid. The wireframe represents only the vertices and edges of a shape. The surface modeling of objects is used to represent surfaces, usually after the user has specified the equation of this surface. Solid modeling represents solid objects, and these objects contain not only geometric information like in surface modeling, but also topological information about the connectivity of the shapes that constitute the solid object.

According to Mantyla, Solid Modeling is a branch of geometric modeling that emphasizes the general applicability of models, and insists on creating only "complete" representations of physical solid objects, i.e. representations that are adequate for answering arbitrary geometric questions algorithmically (without the help of interaction with a human user) [16].

There are three different kind of solid models:

- *Decomposition models:*

  This type of models create a point set (i.e. a solid object) as a collection of other primitive objects like small rectangles or cuboids etc., using an operation that connects these primitives objects ("gluing" operation). Every object can be created as a collection of cells. The cells have no intersection between them, but their boundaries just touch each other. So each object can be decomposed into a set of neighboring cells.

- *Constructive models:*

  These models create a point set from other primitive point sets, where each primitive set is instantiated from a primitive solid type. These models have more general connectivity methods, than the decomposition models.

  A particular modeler of this type is called constructive solid geometry modeler (CSG). A CSG modeler uses parametrized instances of solid primitive objects. The user can perform Boolean operations on the primitive objects and create every other object from these operations. So every object is a combination of simple primitives. All the primitives that constitute an object can be arranged in a binary tree, where each leaf of the tree is one primitive object and each node is an operation.

  The advantages of this type of modelers are:

  - the solid objects that are constructed from a CSG modeler are always valid objects, because their surface is closed and orientable and this surface encloses a volume and
  - these objects are easy to construct and to modify and their data size is small.

  The primary disadvantage of this type of modelers is the computational procedure for the creation and interrogation of an object is time consuming. In addition, the fact that there are only Boolean operations, limits the flexibility of the modeler for creating and editing the solid objects.

- *Boundary models:*

  These models represent a point set as a collection of its boundaries. If the initial point set is a three dimensional one then its boundaries are faces. Each face has edges as boundaries, and each edge has vertices. So every solid object is represented as a collection of faces, edges and vertices. Its surface has an orientation, which allows the distinction between the interior and exterior of an object's volume. The advantages of a boundary representation model are that:

- the operations that can be performed are considerably more diverse than they are in the constructive models,
- the computational procedures for the creation and interrogation of an object are very fast, and
- the topological relationships between the various boundary elements of the object can be obtained very easily.

The disadvantages of this type of modeler are:

- the objects represented may be invalid,
- the memory occupied by the objects is considerably more than the memory occupied by the constructive modeler objects, and
- the programming code and the structure of the data is complex.

Solid modeling ensures that the objects will be closed and bounded. Therefore the outside of a volume of a solid object can be distinguished from the inside and also mass properties can be determined. Also solid modeling allows the creation and manipulation of complete solid shapes, while the integrity of their representation is maintained [9].

2.4 Selective Geometric Complexes (SGC)

The SGC solid modeling method is based on cells. A cell is a connected open subset of an extent. A detailed description of the definition of extents and cells can be found in [21]. In SGC, a geometric complex is a finite collection of mutually disjoined iD cells and selective geometric complexes are used to create nD sets of points. The cells can represent faces, edges and vertices and the topological information between the cells is easily stored in an incidence graph. The cells can be set to be active or non active and this allows the creation of sets of points which may be open or unbounded or non homogeneous. Also Boolean and set-theoretic operations (closure, interior, boundary) can be performed, based on "subdivision", "selection", "simplification" operators [21].

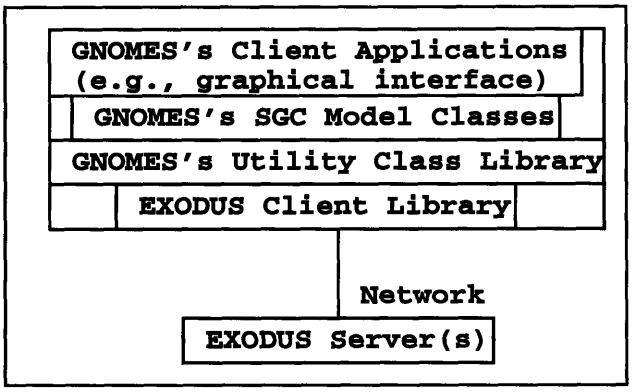

Figure 2-1: GNOMES Architecture

2.5 GNOMES Geometric Modeler

The GNOMES geometric modeler is based on the Selective Geometric Complexes (SGC) paradigm. It is a combination of CSG and boundary representation modelers. It is non two manifold, which means that open, non-homogeneous point sets can be created. Also it has been developed using the object oriented approach. The code is written in the C++ language.

Initially GNOMES was built on a commercial object-oriented database management system ObjectStore (OODBMS) [14]. Objectstore provides many facilities for storing, retrieving objects and asking information about them. Recently GNOMES was modified to work on top of the EXODUS database management system [20],[1], which has similar facilities as ObjectStore.

The GNOMES architecture is shown in Figure 2-1. This figure has been taken from [8], but instead of ObjectStore the EXODUS database is the base layer.

The utility class layer is above the database layer, and it contains classes for performing vector and matrix operations, recording user identity, time stamping, access history, and dynamic definition of attributes. The next level is the SGC model,

which contains all the classes for performing the object operations. These object operations may include low-level operators for creation, traversal, and deletion of the topological data structure, operations on the geometric representation, and high level modeling facilities such as parametric specification of various primitives, sweeping operations, and Boolean set operations [8]. The top level is the user interface. The hierarchical tree of the classes is contained in Appendix A. A more detailed description of the GNOMES solid modeler can be found in [8].

2.6 Features

Many different definitions have been presented for the concept of features on a solid modeler. Some of them are summarized in [24]: "A feature is an entity used in reasoning about the design, engineering, or manufacturing of a product", "A feature is a region of interest", "A feature is a collection (set) of faces of a boundary model". In [25] a feature is defined as a collection of information sets where the sets have been grouped based on the meaning of the information and it is mentioned that features can be informally defined as "recurring patterns of information related to a part's description".

Researchers in Mechanical Engineering have used the term features for describing representation schemes of objects that the computer could use to reason intelligently about their geometry. So features can be considered as higher level modeling elements, compared to the low level primitive geometric elements that were used until now as the building blocks of objects. Features would relate more directly to the engineering domain that they address. Each area of engineering has its own feature types that constitute a taxonomy. In Ocean and Civil Engineering such a taxonomy would include feature types like beams, plates, shells, etc. According to Mantyla, [17], "The success of feature based modeling techniques is largely determined by whether a useful taxonomy of feature types can be identified and organized in a modeling system, and whether application-oriented data and knowledge bases can be conveniently organized on the basis of this taxonomy".

There are different type of features. Some of them are the form features which are volumetric shapes, the precision features which are acceptable deviations from the nominal geometry and the material features which specify material types, grades, properties, heat treatment etc. [25].

Rogers and Shah in [25] extend the definition of features, in order to include their usefulness, as "generic shapes with which engineers associate certain properties or attributes and knowledge useful in reasoning about the product". Features can perhaps be thought of as engineering primitives suited to some engineering task. Also [24] introduces the concept of abstract features, which are defined "as entities that cannot be evaluated or physically realized until all variables have been specified or derived from the model". So the evaluation of these features can be done only when all the information is known. This decreases the flexibility, since all the knowledge has to be acquired in order to evaluate the feature. So a feature would be evaluated only at the very end of the design. Thus, features that could be evaluated partially before design finishes, would be very useful. Features depend on the type of the product they are related, the type of application and the level of abstraction, and a combination of these three factors defines a domain called feature-space ([24]).

2.7 Engineering Design Process

Most of the engineering design processes use the same procedure and steps for accomplishing a design. Three major categories of design can be distinguished ([17]):

- *functional design*

  The designer specifies the desired engineering product. At this stage there is no notion of geometry of the product with the exception of general restrictions about the size, volume and weight, but there are issues about marketability, competitive strategy, etc.

- *conceptual design*

The designer selects a collection of components and subassemblies and defines their relationship, such that the engineering product will meet all the requirements of the functional design.

This is the stage where a designer starts at a top level of abstract geometry of the product, and creates subsystems, until the level of the basic building blocks or primitive components is reached. This concept of abstract geometry is the issue that is mainly addressed by this thesis. The next chapter contains a more detailed description of the abstract geometry, the levels of abstraction and the gradual shape evolution that can be achieved through a computer program.

- *detailed design*

  The main focus is on the detailed design of each of the components and subassemblies that were defined in the conceptual design. All the restrictions and requirements that are imposed by the engineering theories are applied and the detailed geometric dimensions and properties of each component are calculated.

Nevertheless, these are only the main categories and each design process may include many different subcategories, based on the type of design.

2.8 Existing Systems - Literature Survey

Many of the existing systems today use the concepts of feature modeling and they use different feature types for the representation of objects. Pratt and Wilson, whose research is reviewed in [25], defined a hierarchy of features types, where the classification of these types was based on two different categories, the evaluated and unevaluated geometry. Other researchers like Shah, Dixon, Simmons etc., whose research is also reviewed in [25], have developed several other feature-based applications. Also several studies have been performed that support dimensional tolerancing. Further, features have been used extensively for systems that support machining operations [2].

Smithers in reference [26] points out that geometry-based systems cannot be very powerful to support the engineering design. Also he mentions that AI-based systems

cannot support by themselves the process of designing. In the same paper he presents the concept of activity-based model, which involves "the construction of a hierarhical representation which explicitly contains descriptions of the types of features (and possible subfeatures) which constitute the design and how they are functionally and spatially related, together with the description of their geometry."

Laako and Mantyla in reference [13], describe the creation of a feature definition language and a feature-based modeling system, developed using the object oriented paradigm. The feature classes of this system are organized into a taxonomy, which allows the inheritance of the common information from the parent classes to the subclasses.

Woodbury and Oppenheim in reference [31] describe an approach to the integration of geometric information in knowledge-based CAD systems. According to their approach, four concepts should be used as the basis for a CAD system:

- classes of spatial sets,
- features,
- abstractions, and
- constraints.

Mantyla in reference [17] refers to the development of a CAD system that allows top-down design. In the same reference he states that "the design system should provide facilities for modeling the designed object on a purely abstract level - as a structure of model entities, their relationships, and their properties." Further, he mentions that the design system should allow the user to choose the level of detail of the representation.

In reference [30], Wong and Sriram mention that a geometric modeling system that would support cooperative product development should allow the representation through multiple levels of abstraction and multiple viewpoints of cooperating members. Also they mention that "the system should allow crude spatial details, overall shapes, or envelopes to be associated with these levels of abstraction."

From the literature survey performed, it is clear that many authors have addressed the need for the concept of different levels of geometric abstractions. Also some other authors describe classification of objects for use in different procedures, like machining, tolerancing, etc. However, no system was found that combines a taxonomy basis with evolving shape description through different levels of abstraction. Our system addresses the issue of object classification for each separate engineering discipline and further it performs this classification into a hierarchy that allows evolving shape description through abstract levels. In addition, the classes of the system embed knowledge that smoothens the transition between the various levels of abstraction. So each general level of abstraction contains different sublevels, organized in a rule-based manner. Also each class of the GAB module can be considered a feature type, because it can reason intelligently about the geometry of the object it represents, it is directly related to the engineering domain, and it is a higher level modeling element.

2.9 Requirements for the GAB Module

The requirements for our system are:

- to allow the creation of common engineering objects used in Ocean and Civil Engineering, like beams, plates, shells, etc.
- to allow the gradual shape evolution of the engineering objects through different levels of abstraction, based on the knowledge of the type of object that it will be used.
- to allow gradual shape evolution of the engineering objects inside each level of abstraction, based on the available dimensions.
- to enable display based on a concept of "minimal information" required to represent an object.
- to allow a full range of geometric operations.

- to provide the designer with the facility to store user defined parametrized objects, and re-create them each time by providing only the parameters.
- to provide a generic interface which serves as a communication module with client applications (intelligent design agents, reasoning systems, functional modelers).

2.10 Chapter Summary

In this chapter, we introduced some of the basic concepts and techniques that we will use in this thesis. Our objective is to study the issue of knowledge representation for geometry, with a focus on conceptual design. We have discussed the geometric aspects, as in geometric modeling research, and more abstract representation issues, as represented by features. We will use object-oriented programming as an implementation paradigm to allow for shape representation and evolution through the different levels of abstraction in design.

Chapter 3

Geometric Abstractions

3.1 Chapter Overview

This chapter addresses the concept of levels of abstraction. The purpose of the levels of geometric abstraction in this thesis is to support the conceptual design stage. At this stage, the designer does not know all the information for the object being designed, and he/she needs an abstract representation of it.

The specific domain focus of this thesis for the geometry representation is in the areas of Ocean and Civil Engineering. The classification of the objects into a taxonomy, will relate the primitive building blocks directly to the domain they seek to model. The objects are classified into line-forming elements and surface-forming elements. Line-forming elements can be further classified into straight and curved, while surface-forming elements into planar and curved. Also curved-surface elements can be single curved or double curved [23].

The Geometric Abstractions or GAB module is a collection of spatial classes that form a layer of abstraction over the actual geometry representation. GAB supports conceptual design, through the use of the different levels of abstraction. The GAB taxonomy represent objects commonly used in the areas of Ocean and Civil Engineering, such as beams, plates, shells, etc. The object representation can be gradually updated based on the level of the information acquired during the evolution of design. GAB uses two different levels of abstraction, internal and external. The external ab-

straction is between the classes and it is supported by the concept of inheritance of object oriented programming. The internal abstractions are provided by the knowledge contained inside each class. Each class has the knowledge to display a partial representation of the object, based on the information stored in the attributes of the class.

Further, GAB allows the user to parametrize an object and thus create user-defined shapes.

3.2 The Concept of Geometric Abstractions

This chapter addresses the notion of abstraction as a complexity reducing mechanism for the representation of geometry.

Human designers usually tend to use multiple representations for an object depending on the stage of the design, the available information, and the current focus. Most CAD systems use the concept of two manifold solid modeling, which makes the assumption of closed point sets. The geometric modeler used for this thesis is the non-two manifold GNOMES solid modeler, which allows the representation of mixed dimensional point sets. This means that the GNOMES system can model surface, solid and wireframe objects together with non-two manifold conditions in a unified framework of data structures. This representation scheme is structured into a set of geometric abstractions that can be used directly by application programs and this allows the creation of a unified framework for multiple levels of abstraction.

"An abstraction is a partial model of a geometric object," according to Woodbury and Oppenheim [31]. It is further mentioned that abstractions are simplification mechanisms, that allow the expression of a geometric model in a simpler manner.

Woodbury and Oppenheim [31] further mention that abstractions have four properties:

- they constitute partial models of objects
- each abstraction has three parts:

- some data that constitute the model of the abstraction
- a label, that provides a name for the abstraction
- a reference which points to the abstracted object

- they define hierarchies of abstractions, and
- they are automatically maintained under change.

In the GAB module the abstractions have been considered in two different ways. The first is abstractions between the classes and they are performed because of the class hierarchy and the second one is abstractions inside each class. We call them external and internal abstractions.

The purpose of the geometric abstractions is to support the conceptual design stage. At the initial stages of design when the user does not know all the information about the objects, he/she needs a partial representation of it. This helps the design process, because the designers can display on the screen the objects that they have designed and correct inconsistencies and discrepancies from the initial stages.

3.3 Object Classification

The creation of a domain taxonomy involves the following issues:

- decide on the objects mainly used in the areas of Ocean and Civil Engineering, and
- decide on the hierarchy of these objects in order to acquire a basis for the creation of the different levels of abstraction.

In this work, objects are classified into line-forming elements and surface-forming elements, following Schodek [23]. Also line-forming elements can be further classified into straight and curved and surface-forming elements into planar or curved. The curved-surface elements may be of single or double curvature. In reality there is no object with zero thickness, but for the purpose of creating different levels of

abstraction it is very useful, because it allows the classification of all the objects in a particular hierarchical manner, and identifies shape information of primary and secondary importance in the design.

In GAB module the hierarchy of the classes was based on the object classification mentioned in the above paragraph. At the top level there is the class **Engineering_object**, which is the root class and provides many facilities for all the other classes. At the next level there is the straight line, the curved line, the plane surface and the curved surface. All the other objects that are widely used in Ocean and Civil Engineering can be derived from these top level objects. For example all the beams can be represented as a straight line at the initial stages of design, and then, when the exact cross section is known, more information can be added to the straight line. Also, a pipe can be represented as a straight line at the initial stages and then the cross section can be added at an intermediate stage. Similarly a plate or a shell initially can be a two dimensional surface and then the thickness may be added. Also at the top level after the root class, there exist all the other primitive shapes: cuboid, cone, cylinder, circle, rectangle, and sphere. From these primitive shapes, arbitrary objects can be derived by performing appropriate Boolean operations.

3.4 General Description of GAB

The GAB module is a collection of spatial classes corresponding to commonly used objects in the domain of Ocean and Civil Engineering. These spatial classes constitute a layer of abstraction over the actual geometry representation, which in our case is based on the GNOMES non-two manifold geometric modeler. The facility for the incremental specification is based on the concept of inheritance of the object oriented programming together with the evolving object geometry of the taxonomy.

The hierarchical structure of GAB, which allows the incremental specification and the evolution of object geometry, is shown in Figure 3-1.

Figure 3-2 contains the architecture of the GAB module.

GAB module provides the designer with many different capabilities for the cre-

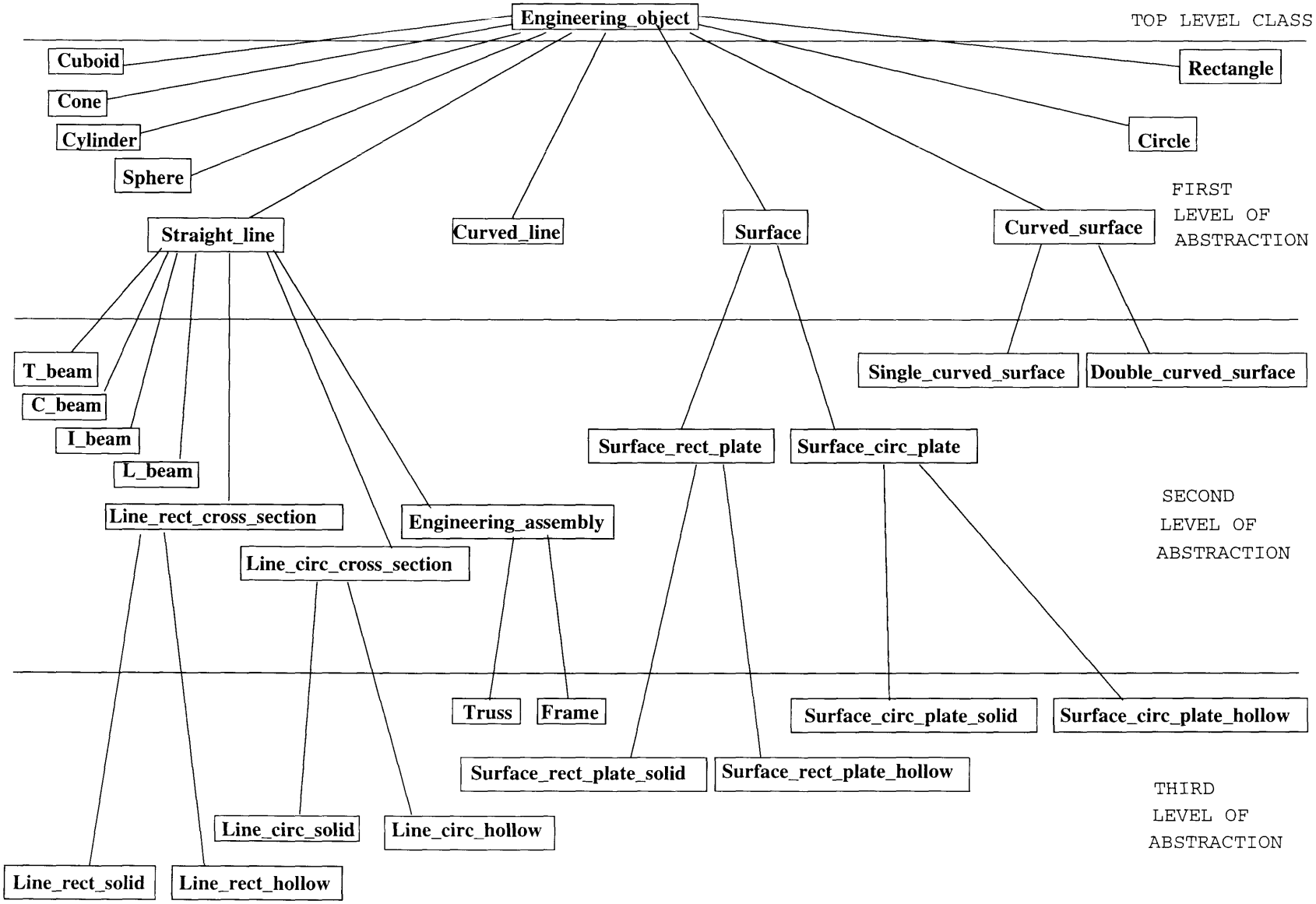

Figure 3-1: GAB Hierarchical Tree

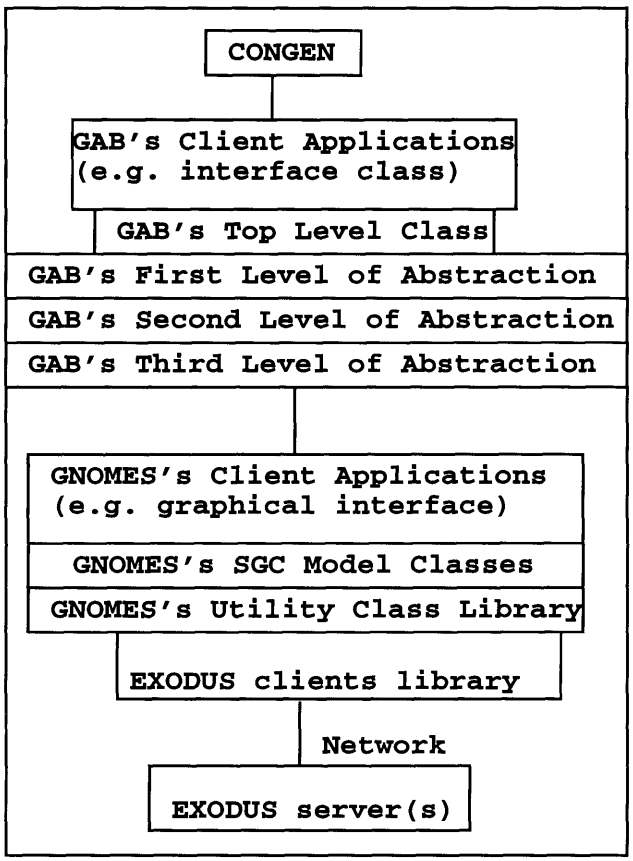

Figure 3-2: GAB Architecture

ation of objects. First if the user knows all the information about an object, he/she can create it directly. If the exact type of object is not known from the initial stages of design, then the user can create an abstraction of it through the hierarchical subdivision of the classes. Then as knowledge is gradually obtained, the designer can upgrade the object and create a more detailed description of it. This is what we call external levels of abstraction. If the type of object is known but all the dimensions of it have not been calculated yet, then an abstraction of the object can be displayed by the knowledge provided inside each class. This is what we call internal level of abstraction. The internal level of abstraction provides the system with the knowledge to represent the engineering objects either in full detail or in a simpler manner, according to existing information.

Further, the GAB module has the capability of handling assemblies. An assembly is considered to be a collection of engineering objects, where some Boolean operations may have been performed on these objects. The root class of the GAB module has the capability of storing assemblies. Also the root class provides methods for performing Boolean operations and creating assemblies.

The GAB module provides the user with the capability of creating user defined objects. This is done through what we call the "parametrization of an engineering object". The concept of the parametrized object in GAB allows the module to "learn" the creation of new objects. This is because the user creates an object for the first time and then it passes this object to the appropriate class. So the user "shows" to the class the method for creating this object. Then he/she is asked for the constant and variable dimensions and for a name. Finally this object is stored under the list of objects. This list of objects is available to all the users that use the same database. So the list of objects grows each time a new object is inserted in the list.

Finally a generic class is provided to control all the capabilities of the GAB module. This class acts as a link of GAB with the domain, which means that it is the interface of GAB with other modules. The modules that currently use the GAB module are:

- CONGEN, a domain-independent knowledge-based design support system, which supports symbolic evolution of design [7], and

- a constraint manager which allows the determination of alternative geometric configurations during each stage of design (currently under development at the Intelligent Engineering Systems Laboratory (IESL) of the Civil and Environmental Engineering Department of MIT).

3.5 The Concept of Abstractions in GAB

Two different types of abstractions have been considered. The first type is abstraction provided from the class hierarchy which we call external abstraction, because it is performed through the class hierarchy. We call the second type of abstraction internal, because it is performed inside each class.

If the exact type of the object that will be used for the design is not known, then an abstraction object can be created. For example if the user wants to create a beam but the exact type of beam is not known, then he/she can create a straight line at the position of the beam, or create a bounding box. In the GAB module, the straight line is one level of abstraction higher than the beams. So this abstraction is external, between classes and it is used when the exact type of object is not known. The knowledge for the external abstractions is contained in the hierarchical structure of the classes. For example all the beams are children of the straight line. When the application acquires the additional information about the type of the object, then it can perform shape evolution, by creating a new upgraded object, through the special constructor that exists in every class. This constructor takes care to assign to the upgraded object, the attributes of the old one.

When the type of the object is known, but the dimensions of the object are not completely defined, then internal abstractions are used. The knowledge for performing the internal abstractions is contained within the methods of each class, in the form of rules. For example if the designer knows that a T beam should be created, but does not know all the dimensions, then an abstraction of the T beam is created but this abstraction is handled by the T beam class internally. When additional information is obtained, the same T beam is recreated including the new information.

3.6 Chapter Summary

The purpose of the GAB module is to support the engineering design process at the initial stage of conceptual design. Then the engineer does not know all the information about the object being designed and he/she requires an abstract representation of it. The support of the conceptual design can be achieved by the concept of the levels of abstraction, which allow the user to evolve the object shape through successive geometry steps.

The domain of this thesis is the engineering objects in the disciplines of Ocean and Civil Engineering. The objects are classified into line-forming and surface-forming elements. Further, the line-forming elements can be classified into straight or curved and the surface-forming elements into planar or curved.

The class hierarchy of the GAB module was based on the above mentioned object classification. Each class represents a family of engineering objects. GAB allows the user to perform evolving shape description through various levels of abstraction. There are two different kinds of abstraction in the GAB module. We call the first external abstractions and they allow the user to evolve a shape from the initial stages of design when the exact geometry is not known, to the very end of the design, when complete knowledge of the geometry exists. This is performed through the concept of inheritance of the object oriented paradigm. The second type of abstractions was given the name internal abstractions, because it is performed inside each class. The internal abstractions are used when the exact type of object is known, but all the features of this object have not been acquired. This type of abstraction uses embedded knowledge in the classes .

Further, the GAB module allows the designer to define his/her own objects, by parametrizing them. Finally a wrapper class was designed that serves as a reference to the geometry, contains domain knowledge about default values for attributes, and interaction interfaces with various client applications.

Chapter 4

Detailed Description of Geometric Abstractions Module (GAB)

4.1 Chapter Overview

This chapter provides a detailed description of the GAB classes. Section 4.2 gives a generic description of the purpose of each class and the knowledge included in it. Section 4.3 provides a description of the wrapper class which acts as the link of the GAB module with the client applications. This is followed by a detailed description of the attributes and methods of each class in Section 4.4. Many methods are virtual and they provide a polymorphic interface for response to messages. In this chapter we use the word "user" to mean either a human designer or a client application such as an intelligent design agent. At the end there are examples that show the use of the GAB capabilities.

4.2 Description of GAB Classes

The GAB module is written in the C++ language, using the object oriented paradigm. GAB is not a solid modeling engine, but it is built on top of the GNOMES non-two manifold geometric modeler, and it uses the capabilities of GNOMES in order to create the objects. The classes of the GAB module are database classes, so all the

information they contain is stored in the database. Each class represents one object and contains all the information required for the creation and manipulation of this object. The header files for all the classes are contained in Appendix B.

All classes have the same architecture with respect to the methods they include, with the exception of the top level class **Engineering_object**. Since they have similar architecture, all the common methods will be described at the beginning and the description will not be repeated for each class separately.

At first, all the classes contain four constructors, with the exception of the top level. The first constructor is for the creation of a new object, without any previous information and without evolution from another object through levels of abstraction. This constructor takes as argument the object name. The second constructor is for the evolution of an object to the next level of abstraction and it takes as arguments the name of the new object and the address of the previous one that will be upgraded. All the attribute values of the abstracted object are copied to the new one. The third constructor is for creating an object from one that is at the same level of abstraction and in this case the attributes of the initial object are copied to the new one. The fourth constructor creates a new object, by taking as arguments, values for all the attributes of the object.

The method *propagate_attribute_values* has the knowledge to check which attributes have zero value, and assign a value to them, if one can be found for example from the bounding box. This method utilizes knowledge encapsulated in the form of rules.

The method *check_attribute_values* encapsulates the knowledge about the minimal information required to display an object at each level.

The *display* method computes and displays the object at the best possible level of detail. The knowledge to do this is captured in the form of heuristics.

- **Engineering_object:**

  It is the top level class. It contains information common for each engineering object. The attributes of this class are very generic attributes that all objects have and inherit. Also it provides the generic virtual interface for all classes. In

addition to that, it contains information about a bounding box, which will fully contain each engineering object. Further it provides operators for the union, difference and intersection of engineering objects. These operators perform the Boolean operations between the solid models that represent each engineering object. These solid models are stored in the class **Engineering_object** under the attribute **GNmodel* model**. The type **GNmodel** is a type of the GNOMES non-two manifold geometric engine and represents a geometric model. The results of the Boolean operations are non standard **Engineering_objects**, but they are stored as **GNmodels** of an **Engineering_object**. So the class **Engineering_object** can handle assemblies that have been created from Boolean operations between other **Engineering_objects**.

- Primitive object classes: **Cuboid, Cone, Cylinder, Circle, Rectangle, Sphere**

  They are subclassed from the **Engineering_object** and they create the primitive objects cuboid, cone, cylinder, circle, rectangle, sphere. They are created by directly using the facilities of GNOMES.

- **Straight_line:**

  This is a subclass of **Engineering_object** and it creates a line. For a line the attributes area, volume, weight, which are inherited from the parent class, are zero but since this class is parent for many other 3D object classes, these attributes are inherited to the children classes.

  At the initial stages of designing a beam, pipe, column or truss, the designer may not know the exact type of object that he/she will use. So the representation of these objects as a line is the initial level of abstraction. If the user does not specify a value for the axis, then the straight line will be created by default on the x axis.

  The line is one of the primitive objects of the GNOMES geometric engine, and in GAB it is created by directly calling the appropriate method of GNOMES.

- **Line_rect_cross_section:**

  This class creates a straight line with rectangular cross section. It is a subclass of **Straight_line**. In other words this class creates a beam or any other long, beam-like object that has a rectangular cross section. But the rectangular cross section might be hollow or solid. So this class provides a step between the straight line and a beam of full detail. This object is represented in wireframe, because it is still not known if it will be a solid or a hollow beam. Therefore this class is the parent of a solid and a hollow rectangular beam. If the width or the height of the cross section is not known, then a two dimensional surface is created. If both of them are unknown, then a straight line is created. If the user does not specify a value for the axis, then the object is created on the x axis.

- **Line_circ_cross_section:**

  This class creates a straight line with circular cross section. It is a subclass of **Straight_line**. In other words this class creates a beam, or a pipe or any other pipe-like object with a circular cross section. But the cross section might be hollow (for example for a pipe) or solid (for example for a circular beam). So this class provides a middle level of abstraction between the straight line and a full-detailed object of circular cross section. It is represented in wireframe, because it is not still known if it will be solid or hollow.

  If the radius of the cross section is not known, then the class has the knowledge to create a straight line. The x axis is the default axis when the user does not specify a value for it.

  This object is created by calling the appropriate method of GNOMES for the creation of a cylinder.

- **Line_rect_solid:**

  It is a subclass of **Line_rect_cross_section**. This class represents the last level of abstraction. It creates a straight line of solid rectangular cross section in

full detail. So this class is assumed to be called at the end of the design or whenever the designer knows all the geometric information about this object. But even if the user calls this class at the initial stages of the design without providing all the geometric dimensions for the cross section, then this class has the knowledge to represent an abstraction of the object. If the length is not known, then nothing will be displayed. This object is created as a GNOMES cuboid.

- **Line_rect_hollow:**

  It is a subclass of **Line_rect_cross_section.** It represents the last level of abstraction for a straight line with hollow rectangular cross section. If all the dimensions of the object are defined with the exception of the thickness, then it is created with no thickness, which means that there are four long and narrow surfaces that create a long hollow box. If the width or height of the cross section is not known, then a straight line is created, otherwise when the length is unknown no object is created. The x axis is the default axis.

  This object is created as the difference of two GNOMES cuboids.

- **Line_circ_solid:**

  It is a subclass of **Line_circ_cross_section.** This class represents the last level of abstraction. It creates a straight line of solid circular cross section in full detail. So this class is called at the end of the design or whenever the designer knows all the geometric information about this object.

  If the user does not specify a value for the radius, then a straight line is created, but if he/she does not give a value for the length, then nothing is created. This object is created as a GNOMES cylinder.

- **Line_circ_hollow:**

  It is a subclass of **Line_circ_cross_section.** It represents the last level of abstraction for a straight line with hollow circular cross section. If the thickness of the hollow object is not known, then the system assigns a small value of:

radius/40 to it. If the length is not known, then nothing is created. Also if the user has not specified the axis on which to create the object, then the x axis is taken as default. This object is created as the difference of two GNOMES cylinders.

- **T_beam:**

  This class creates a T beam. It is a subclass of **Straight_line**. The T beam is created as the union of two GNOMES cuboids. If one of the dimensions is not known, then this class has the knowledge to construct an abstraction of the T beam. If the total width or total height of the cross section are not known, then a straight line is constructed. If the head thickness is unknown, then the T beam is constructed as the union of a cuboid for the foot section and a two dimensional surface for the top section. If the foot thickness is not known, then the T beam is constructed as the union of a cuboid for the top section and a two dimensional surface for the foot section of the beam. When all the dimensions are known, then the beam is constructed as the union of two GNOMES cuboids. Also the default axis is the x axis.

- **C_beam:**

  This class creates a C beam. It is subclassed from Straight_line. It is created as the union of three cuboids. It has the knowledge to construct an abstraction of the C beam. If the thickness is not defined, then 2D surfaces are constructed. Also if the total width or height is unknown, then a straight line is created and finally if no axis is specified, then it is created by default on the x axis.

- **I_beam:**

  This class creates the I type of beam. It is also subclassed from the straight line as all the other beam-classees. When all the dimensions are defined, the I beam is created as the union of three cuboids. But if some dimensions are not specified, then this class has the knowledge to display some abstractions of the I beam, for example 2D surfaces instead of cuboids, or a straight line, if the

total height or width of the cross section is not known. The default axis for the creation of the I beam is the x axis, as in the case of all the previous beams.

- **L_beam:**

  The L beam is the last type of beam that can be created in GAB. The addition of the L beam, provides the designer with the most commonly used types of beams in the Ocean and Civil Engineering.

  This class has the knowledge to create a full detailed object or an abstraction of it, based on the available information.

- **Curved_line:**

  The curved line is a subclass of **Engineering_object** and it creates an arc. The user has to specify the angle and radius of the arc. If one of the two is not known, then nothing is created or displayed. But if the user knows the chord height or the length of the arc instead of the radius or the angle, then he/she can specify it and the class has the knowledge to calculate the radius and angle. There is no internal abstraction for this class. This means that if the angle and radius are not known or cannot be calculated, then nothing is created.

  GAB is built on top of the GNOMES non-two manifold geometric engine, and therefore the curved line is created as an arc of GNOMES. In GNOMES the arc is always positioned at the x-y plane with its center at the beginning of the axis. The starting point of the arc is at the x-axis, at a distance from the beginning of the axequal to the radius. Since GNOMES does not allow the users to specify the plane and the center of the arc, also GAB does not allow them to define any plane or center for the curved line. However, after the creation of the curved line, the users can translate or rotate it to the desired position.

- **Curved_surface:**

  The curved surface class is a subclass of **Engineering_object.** It represents a curved surface as a part of a cylindrical surface patch which is created by taking the difference of a cylinder and a cuboid. It requires the knowledge of

the radius, width and angle of the curved surface. The radius and width of this curved surface correspond to the radius and height of the initial cylinder. The angle represents an angle on a cross section of the cylinder and defines which part of the cylindrical surface patch will be taken. The cylindrical surface patch is created around the x-axis, half at the positive side and half at the negative side. The height of the surface patch is towards the positive side of the z-axis.

If the width is not known, then an arc is created, as the projection of the cylindrical surface patch on the y-z plane.

This class contains the knowledge to calculate the angle and radius from the chord height and length of the arched surface. The formulas used are the same with the previous case of the curved line. If the angle or radius are not known or they cannot be calculated then no minimal representation of the curved surface can be created. Appendix C contains a drawing of the top and side view of the position that the curved surface is created.

- **Single_curved_surface:**

  This class is the next level of abstraction after the curved surface. All the information of the parent class is inherited and the only knowledge added is the thickness of the single curved surface. The object represented by this class is a single curved shell. It is created around the x-axis, half at the positive side and half at the negative side and towards the positive side of the z-axis. Appendix C contains a drawing of the top and side view of the single curved surface.

  If the thickness is not known, then a curved surface without thickness is constructed.

  This object is created by taking the difference of two cylinders and then by subtracting a cuboid. Figure 4-1 shows the two cylinders and the cuboid.

  Usually in the disciplines of Ocean and Civil Engineering the curved plates and surfaces have their edges perpendicular to the surface. Currently the edges of the single curved surface are not created perpendicular to the surface, as shown

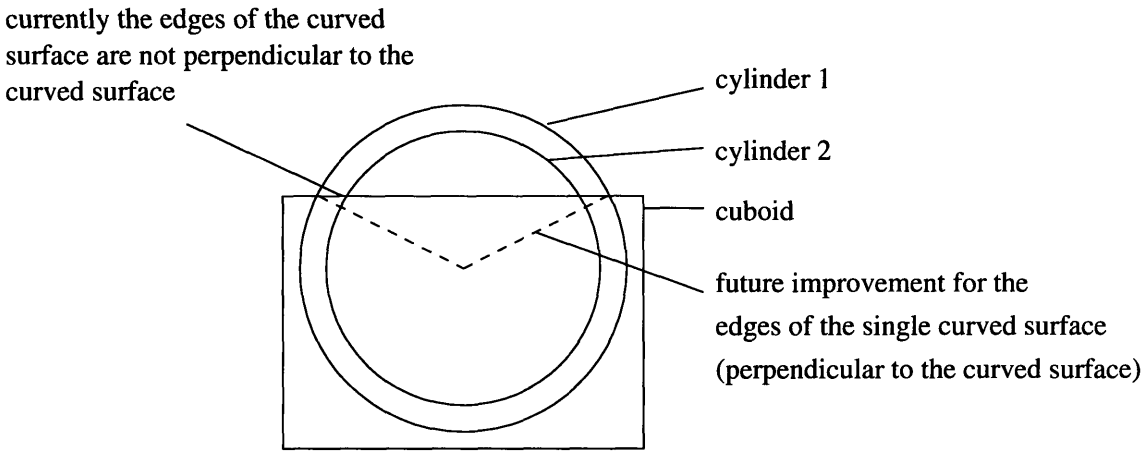

Figure 4-1: Single Curved Surface Creation

in figure 4-1. A future improvement will create the single curved surface as the difference of other objects that will allow the edges to be perpendicular to the surface.

If the user specifies a very small angle, then the height of the curved surface may be less than the thickness, as shown in figure 4-2. In this case the object created as the difference of the two cylinders and the cuboid is not the desired one. However this class has the knowledge to avoid the creation of such an object, and it informs the user that a planar surface should be created instead. Since the angle is very small, the curved surface can be approximated by a planar plate.

- **Double_curved_surface:**

The double curved surface is a subclass of the curved surface. It is created as the difference of two spheres and from this difference a cuboid is subtracted and therefore this object is a shell with thickness. The input data required are the radius, the angle and the thickness. The radius represents the radius of

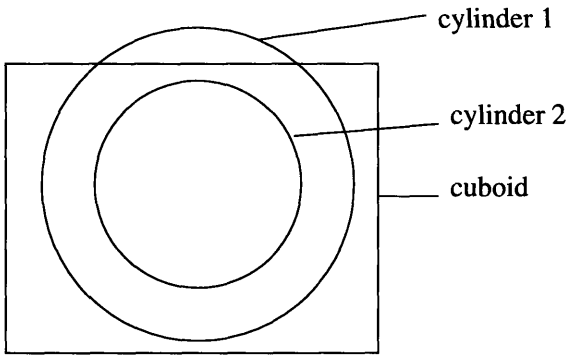

Figure 4-2: An Undesired Case

the larger sphere, while the thickness is equal to the difference of the radii of the two spheres. The angle represents an angle on a cross section of the sphere and defines which part of the spherical surface will be taken. The radius and the angle are assumed to have the same value in the two axes x and y. All the information of the parent class **Curved_surface** is inherited to this class. While the **Curved_surface** has a width, the double curved surface object does not need to use the width, because this will be equal to the radius. Nevetheless if the width is specified and no radius is given, then the width will be assumed to be equal to the radius. Else if both the width and radius are defined with different values, then the radius is assumed to be the valid one. Appendix C contains a drawing of the top and side view of the double curved surface.

The edges of the double curved surface are not created perpendicular to the surface, as in the case of the single curved object. So a future improvement needs to take provision for creating the edges perpendicular to the surface. Similarly with the case of the previous class described, when the angle specified is very small, this class has the knowledge to avoid the creation of an undesired

object and informs the user that he/she should create a circular plate instead.

The minimal representation of this object requires the knowledge of both the radius and the angle.

- **Surface:**

  This class creates a 2D planar surface. It is a child of the **Engineering_object** class. The object is created as a rectangular parallelogram and it represents a 2D plate. The object is created in the x-y plane by default, but the user can specify another plane if desired.

- **Surface_rect_plate:**

  The **Surface_rect_plate** class is a subclass of the Surface class. It creates a rectangular surface with thickness. This object is represented by a cuboid, but since it can be either solid or hollow, the object is represented in wireframe form. The children of the current class, have the knowledge to represent either a hollow or a solid rectangular plate. If the thickness is not specified, then the object is created as a 2D rectangular surface. The default plane is the x-y.

- **Surface_circ_plate:**

  The **Surface_circ_plate** is a subclass of the Surface class. It creates a circular plate with thickness. This plate can be either hollow or solid. The children classes have the knowledge to create the hollow or solid plate. So the representation of the object in this class is in wireframe mode, since it is not known if the object will be solid or hollow. If the thickness is not known, then a circle is created. This class inherits from its parent class the x and y dimensions of the plate. When an object created through the **Surface** class is evolved to a **Surface_circ_plate**, then the average of the x and y dimensions are assumed to be the radius of the new object.

- **Surface_rect_plate_solid:**

  This class creates a solid rectangular plate and it is subclassed from **Surface_rect_plate**. The internal level of abstraction provides the facility to create

a 2D rectangular plate instead of the 3D one, if the thickness is not defined. The plate is created in the x-y plane by default, unless the designer specifies another plane.

- **Surface_rect_plate_hollow:**

  The **Surface_rect_plate_hollow** is a subclass of **Surface_rect_plate.** This object is a rectangular 3D plate, which is hollow at the inside. This object could be used as a double hull cell for ships.

- **Surface_circ_plate_solid:**

  The solid circular plate is a subclass of **Surface_circ_plate_solid.** While the objects of the parent class are created in wireframe mode, the objects of this class are created solids. With respect to the attribute values, all the information remains the same as the parent class.

- **Surface_circ_plate_hollow:** The solid circular plate class is a subclass of **Surface_circ_plate_hollow.** The objects of this class are hollow cylinders with thickness.

- **Parametrized_object:**

  This class provides the user with the capability of creating user defined objects. The class takes the address of an engineering object or assembly and parametrizes it.

  The first time the user has to create an assembly using the standard objects from the GAB module and also using Boolean operations. Then the designer can pass the address of the assembly to the class **Parametrized_object** and then this class has the knowledge to ask him/her, what dimensions should be constant and what dimensions should be variables. For the constant dimensions the class keeps the values of the initial assembly. Then the user is asked to give a name for this parametrized object and this object is stored under the list of objects. The next time that the user will call this name, he/she will be asked

only for the values of the variables. Then the assembly will be displayed in the position of the initial one, but with the new values for the variables.

This part of the GAB module needs further development. The reason is that the objects that constitute the assembly are created with respect to the center of volume. So the center of volume is placed each time to the same position. This might lead to inconsistent objects, depending on the value of the variable dimensions. What is needed is the addition of constraints to the **Parametrized_object** class. Also it is necessary to create the objects with respect to a constant point on the object that the user will define and not with respect to the center of volume. In this way the invalid objects will be avoided.

4.3 Description of the Wrapper Class Gab_object

The class **Gab_object** is provided to act as the link of GAB with the client applications, which means that it is the interface of GAB with other modules. The modules that currently use GAB are:

- CONGEN, a domain-independent knowledge-based design support system, which supports symbolic evolution of design [7], and
- a constraint manager which allows the determination of alternative geometric configurations during each stage of design (currently under development at the Intelligent Engineering Systems Laboratory (IESL) of the Civil and Environmental Engineering Department of MIT).

Figure 4-3 represents the possible ways of communication between the **Gab_object** class and the two modules: CONGEN and the constraint manager.

The CONGEN module can send messages to GAB for the creation of new objects, or for the allocation of attribute values, or for the evolving shape description of objects. Also the CONGEN module can send messages to the Constraint Manager for the calculation of values of attributes. The Constraint Manager may send messages

1. CONGEN sends messages to the GAB module for the creation or evolving shape description of objects
2. CONGEN sends messages to the constraint manager for the calculation of attributes values
3. The Constraint Manager sends messages to the GAB module in order to assign values to generic attributes or to request information for already set attributes
4. The GAB module replies to the Constraint Manager by sending the values of the already set attributes
5. Returns a reference to the geometry
6. Returns consistent assignment of variables

Figure 4-3: Communication Between the Modules

to the GAB module for acquiring the values of any attributes that are already set or it may set the values of some generic attributes.

This class contains methods for the creation and evolving shape description of the GAB objects. The client applications have to declare a variable of type **Gab_object**, and this object can be created, displayed or upgraded, without changing its address.

Also this class can store the type of object that the client application might want to design, but it has no knowledge to create it. The object is being created by calling the appropriate GAB classes.

4.4 Description of Class Attributes and Methods

All the classes have the same architecture with the exception of the wrapper and top level classes. The wrapper class acts as the GAB's interface to the clients applications and the top level class contains many utility methods for all the other classes. The methods *set_attribute_values*, *display*, *propagate_attribute_values*, *check_attribute_values* are common methods to all classes.

- *virtual void set_attribute_values()*: this method asks the user for the values of all the attributes. Since each class has its own attributes, this method is a virtual one, and it will be evaluated during execution time. Additionally, each class has other methods for assigning values to each attribute separately. The other methods set one attribute at a time. So if the user has partial information about the object, then he/she should use the methods that set each attribute separately. The use of the separate methods for each attribute allow the designer to display an abstract representation of the object, based on the available information. The method *set_attribute_values* would require from the user all the information. Nevertheless if the designer uses this method without knowing all the information, then he/she can give a zero value at the unknown variables and the result would be the same.

- *virtual void display*: this is a virtual method. It contains all the information required for the creation and the display of the object of each class. For the class Engineering_object this method creates the bounding box.

- *virtual void propagate_attribute_values()*: this method is called after the user has assigned values to the attributes. It checks which attributes have a value and if some attributes have zero value, then the method has the knowledge to assign a value if it is possible, from other known information, like the bounding box. For example if the object that the user wants to create is a cone, but he/she has assigned values only for the radius, then if there is a bounding box, the value of the height of the bounding box is automatically assigned to the height of the cone. If there is no bounding box, then nothing is displayed on the screen.

- *virtual int check_attribute_values*: this method is also a virtual one and it is called before the creation of the object. Its purpose is to check if the user has specified enough values for the attributes, that would allow the creation of the object, or if some important data are missing that will not allow the creation of a valid object. If the data are complete then the method "display" has the knowledge to display an abstraction of the object or a complete object, depending on the information available.

In addition to the above methods all the classes have four constructors. The first one is the default constructor. The second one is used for the creation of the object that the class represents, by giving the name of the object. Then the user has to specify the values for the attributes of this object. The third constructor is used for the gradual shape evolution. This constructor copies all the arguments of the object from the previous level of abstraction to the current object. The fourth and last constructor creates the object by taking the values of its attributes. The only exception are the class **Gab_object** and **Engineering_object** which have different constructors.

4.4.1 class Gab_object

The methods and attributes of this class can be called from all the client applications. Since this is an interface class, the attributes may be zero, if the client application does not have any information about the objects being designed.

Attributes:

- ENGINEERING_OBJECT *OBJECT: this attribute stores the address of the engineering object that the client applications will create. Since the **Engineering_object** is the top level class, all the instances of the other GAB classes can be stored under this address. Also the architecture of the GAB module is such that all the public methods that can be called by the client applications are virtual and therefore all the public methods of all the classes can be accessed.

- DBDOUBLE LENGTH: this attribute represents the length of the bounding box.

- DBDOUBLE WIDTH: this is the width of the bounding box.

- DBDOUBLE HEIGHT: this attribute represents the height of the bounding box.

- GNVECTOR ROTATION: the type "GNvector" is a type defined in the GNOMES solid modeler and it stores a vector. So this attribute represents a vector which contains the total rotation of the bounding box with respect to the beginning of the axis.

- GNVECTOR TRANSLATION: this attribute represents a vector that includes the total translation of the bounding box with respect to the beginning of the axis.

- OBJECT_TYPE OBJECT_TYPE: the type "Object_type" is a type defined by the command "enum". The attribute OBJECT_TYPE represents the type of the object that the client application might create. Nevertheless this class just stores the type of object as a generic knowledge and it does not have any notion of geometry.

Methods:

- *Gab_object*: this is the constructor of the class. It initializes all the attributes to zero.
- *void set_length(double)*: this method sets the length of the bounding box.
- *void set_width(double)*: it sets the width of the bounding box.
- *void set_height(double)*: this method sets the height of the bounding box.
- *void set_rotation(GNvector)*: this method sets the total rotation of the bounding box.
- *void set_translation(GNvector)*: this method sets the total translation of the bounding box.
- *void set_object_type(Object_type)*: it sets the type of the object. This information is stored as generic knowledge and there is no notion of geometry.
- *double get_length()*: this method returns the value of the bounding box length to the client application.
- *double get_width()*: it returns the value of the bounding box width.
- *double get_height()*: it returns the value of the bounding box height.
- *double get_rotation()*: this method returns the value of the total bounding box rotation.
- *double get_translation()*: this method returns the value for the total bounding box translation to the client application.
- *Object_type get_object_type()*: it returns the type of the object that the client application might create.
- *void create_object(Object_type,char*)*: this method takes as arguments the type of object that the user wants to create and the name of the object. Then the appropriate GAB class is called and the object is created and stored under the attribute: ENGINEERING_OBJECT *OBJECT.

- *void upgrade()*: this method is used for the shape evolution of an object. It has the knowledge to create the new evolved shape.

4.4.2 class Engineering_object

Attributes:

- GNMODEL *MODEL: the type "GNmodel" is a type of the GNOMES solid modeling engine, that represents a general type of object. In the class Engineering_object, this attribute contains the address of the object that it is created through the GNOMES solid modeler.

- DBCHAR COLOR[50]: this attribute is a character array of size 50. It contains the color of the object.

- DBCHAR MATERIAL[50]: this attribute represents the type of material of the object. It is a character array of size 50 and contains a character string with the name of the material.

- DBCHAR NAME[50]: it stores the name under which the object will be created. It is a character array of size 50, so the name of the object may have up to 50 characters.

- DBDOUBLE AREA: it contains the surface area of the object.

- DBDOUBLE VOLUME: it stores the value for the volume of the created object.

- DBDOUBLE SPECIFIC_WEIGHT: it stores the value for the specific weight of each object.

- DBDOUBLE WEIGHT: it contains the weight of the object that was created. The weight is a function of the specific weight. It is calculated by multiplying the volume by the specific weight. So if no specific weight has been specified, then the weight will be zero.

- GNVECTOR TRANSLATION: the type "GNvector" is a type of the GNOMES solid modeler, and it is used for defining a vector. This attribute stores the position of the center of volume of the object with respect to the global coordinate system. There is one exception, the cuboid, which position is stored with respect to the left, bottom vertex.

- GNVECTOR ROTATION: this attribute stores the total rotation of the object with respect to the global coordinate system.

- DBINT OBJECT_TYPE: it stores the type of object that it is created, for example if it is a cuboid, or a cone etc.

- DBCHAR WIRE_SOLID[70]: this attribute contains the display mode for the object created, wireframe or solid. It is for private use in the class. The display mode is stored as a character string, and therefore this attribute is a character array of size 70.

- GNCOMPLEX *BOUNDING_BOX: the type "GNcomplex" is a type of the GNOMES solid modeler and it is used for creating a complex, where complex is a primitive object. This attribute stores the address of the bounding box, if a bounding box has been created.

- BOUNDING_BOX_X_DIM: this attribute is the length of the bounding box, i.e. it represents the value of the dimension of the bounding box along the x axis.

- BOUNDING_BOX_Y_DIM: this attribute is the width of the bounding box, i.e. it represents the value of the dimension of the bounding box along the y axis.

- BOUNDING_BOX_Z_DIM: this attribute is the height of the bounding box, i.e. it represents the value of the dimension of the bounding box along the z axis.

- FLAG_MODEL_CREATED: this flag is for private use, in the GAB module. Its purpose is to keep track of the creation of an object. For example, if the user has created a T beam, and then he/she decides to change a dimension, then the object is redisplayed automatically, based on the value of this flag.

- GNSET<ENGINEERING_OBJECT*> CONTAINED_OBJECTS: the type "GNset" is a type of GNOMES solid modeler, that stores a list of **Engineering_objects** in the database.

Methods:

- *void set_name(char *)*: it sets the name of the object.
- *void set_color(char *)*: it sets the color of the object.
- *void set_material(char *)*: it sets the type of material of the object.
- *void delete_bounding_box()*: it deletes the bounding box of the object that calls this method.
- *void delete_model*: it deletes the "model" attribute, of the object that calls this method.
- *void set_bounding_box_attributes(double,double,double)*: it assigns values for the bounding box dimensions.
- *GNmodel* get_model()*: it returns the address of the "model" attribute.
- *void assign_model(GNmodel*)*: the purpose of this method is to assign an already existing object to the address of the "model".
- *GNcomplex* get_box()*: this method returns the address of the bounding box.
- *GNvector get_translation()*: it returns the total translation of the object with respect to the beginning of the axes.
- *GNvector get_rotation()*: it returns the total rotation of the object with respect to the beginning of the axes.
- *int get_object_type()*: it returns the type of the object.
- *void translate(GNvector&)*: this method is used for the translation of the engineering objects.

- *void rotate(GNvector&)*: this method is used for the rotation of the engineering objects.

- *void add_to_translation_vector(GNvector&)*: the purpose of *add_to_translation_vector* is to keep the translation vector updated with the total history of translation of the object.

- *void add_to_rotation_vector(GNvector&)*: the purpose of *add_to_rotation_vector* is to keep the rotation vector updated with the total history of rotation of the object.

- *void display_message*: this method is used for displaying messages. All the classes of GAB can access it to display a message.

- *void read_value()*: all the classes of GAB use this method for reading a value from the user.

- *Engineering_object& operator+(Engineering_object&)*: this operator performs the union between two engineering objects.

- *Engineering_object& operator-(Engineering_object&)*: this operator performs the difference between two engineering objects.

- *Engineering_object& operator*(Engineering_object&)*: this operator performs the intersection between two engineering objects.

- *void set_wire_solid*: this method is not intended for public use, but it is used by the different classes for setting an object to wireframe or to solid mode of representation.

In addition to the above mentioned methods, this class has also the methods common to all classes, which are: *display*, *set_attribute_values*, *propagate_attribute_values*, *check_attribute_values*.

4.4.3 class Straight_line

Attributes:

- DBDOUBLE LENGTH: this attribute stores the value for the length of the straight line.
- DBINT AXIS: this attribute stores the axis where the straight line will be created. If no axis is specified then the x axis is taken as the default.

Methods:

- *void set_length(double)*: it sets the length of the straight line.
- *void set_axis(int)*: it sets the axis that the straight line will be created.

In addition to the above mentioned methods, this class has also the methods common to all classes, which are: *display*, *set_attribute_values*, *propagate_attribute_values*, *check_attribute_values*. These methods are described at the begining of this section. Also this class inherits all the attributes and methods of its parent class.

4.4.4 class Line_rect_cross_section

Attributes:

- DBDOUBLE WIDTH: it sets the width of the cross section of the line.
- DBDOUBLE HEIGHT: it stores the height of the cross section.

Methods:

- *void set_width(double)*: this method sets the width of the cross section.
- *void set_height(double)*: this method sets the height of the cross section.

In addition to the above mentioned methods, this class has also the methods common to all classes, which are: *display*, *set_attribute_values*, *propagate_attribute_values*, *check_attribute_values*, and are described at the begining of this section. Also this class has all the attributes and methods of its parent class.

4.4.5 class Line_circ_cross_section

Attributes:

- DBDOUBLE RADIUS: it stores the value for the radius of the circular cross section.
- DBINT NUMBER_OF_DIVISIONS: this attribute stores the value for the number of divisions. The straight line with the circular cross section is created as a long cylinder. This cylinder is created through GNOMES, where the curved surfaces are represented by many planar connected surfaces. So the number of divisions specifies the number of planar surfaces that the cylinder will be approximated by.

Methods:

- *void set_radius(double)*: it sets the radius of the cross section
- *void set_number_of_divisions(double)*: it sets the number of divisions of the cylinder.

4.4.6 class Line_rect_solid

This class contains no additional attributes than its parent class **Line_rect_cross_section**. Also the methods of this class are the general methods described at the beginning of this section. The difference with the parent class, is that this one represents a solid line with rectangular cross section, while the parent represents a wireframe line with rectangular cross section. The reason for having the class **Line_rect_cross_section** is that the next level of abstraction might be either a line with solid cross section or a line with hollow cross section.

4.4.7 class Line_rect_hollow

This class is subclassed from the **Line_rect_cross_section**. The additional attribute is the thickness of the cross section.

Attributes:

- DBDOUBLE THICKNESS: this is the thickness of the cross section of the beam.

Methods:

- *void set_thickness(double)*: this method sets the thickness of the cross section.

4.4.8 class Line_circ_solid

It is subclassed from the class **Line_circ_cross_section**. This class has no additional attributes. The purpose of the **Line_circ_cross_section** is that it might be either a solid or a hollow circular cross section. The **Line_circ_cross_section** is represented in wireframe, while this class is in solid representation.

4.4.9 class Line_circ_hollow

This class is subclassed from the **Line_circ_cross_section**. The additional attribute is the thickness of the cross section.

Attributes:

- DBDOUBLE THICKNESS: this attribute represents the thickness of the cross section.

Methods:

- *void set_thickness(double)*: this method sets the thickness of the cross section.

4.4.10 class T_beam

All the beams are subclasses of the **Straight_line** class. It is assumed that a beam is a straight line with a particular cross section. Appendix C contains drawings with all the dimensions for the beams.

Attributes:

- DBDOUBLE HEIGHT: this is the total height of the cross section.

- DBDOUBLE WIDTH: this is the total width of the cross section.
- DBDOUBLE FOOT_THICKNESS: this attribute stores the value for the thickness of the bottom part of the cross section of the T beam.
- DBDOUBLE HEAD_THICKNESS: this is the thickness of the top part of the cross section of the beam.

Methods:

- *void set_height(double)*: this method allows the user to set the total height of the cross section.
- *void set_width(double)*: this method allows the user to set the total width of the cross section, which is the width of the top part of the T beam.
- *void set_foot_thickness(double)*: it allows the user to set the thickness of the foot part of the cross section.
- *void set_head_thickness(double)*: it allows the user to set the thickness of the top part of the cross section.
- *void set_thickness(double)*: this method allows the user to set the thickness of the cross section.

4.4.11 class C_beam

The C beam is also subclassed from the class Straight_line. Appendix C contains a drawing of the C beam dimensions.

Attributes:

- DBDOUBLE EXTERNAL_HEIGHT: this attribute stores the value of the external height of the C beam.
- DBDOUBLE INTERNAL_HEIGHT: this is the internal height of the cross section.
- DBDOUBLE TOP_WIDTH: this is the width of the top part of the cross section.

- DBDOUBLE BOTTOM_WIDTH: this attribute represents the width of the bottom part of the cross section.

- DBDOUBLE FOOT_THICKNESS: this attribute stores the value of the thickness of the bottom part of the cross section of the C beam.

- DBDOUBLE HEAD_THICKNESS: this is the thickness of the top of the cross section of the beam.

Methods:

- *void set_external_height(double)*: it allows the user to set the external height of the cross section.

- *void set_internal_height(double)*: this method allows the user to set the internal height of the cross section.

- *void set_top_width(double)*: this method allows the user to set the top width of the cross section, which is the width of the top part of the C beam.

- *void set_bottom_width(double)*: it allows the user to set the width of the bottom part of the cross section.

- *void set_thickness(double)*: it allows the user to set the thickness.

4.4.12 class I_beam

The I beam is also subclassed from the class Straight_line. Appendix C contains a drawing with all the dimensions of the I beam.

Attributes:

- DBDOUBLE EXTERNAL_HEIGHT: this attribute stores the value of the external height of the I beam, which is the total height.

- DBDOUBLE INTERNAL_HEIGHT: this is the internal height of the cross section.

- DBDOUBLE TOP_WIDTH: this is the width of the top part of the cross section.

- DBDOUBLE BOTTOM_WIDTH: this is the width of the bottom part of the cross section.

- DBDOUBLE FOOT_THICKNESS: this attribute stores the value for the thickness of the bottom part of the cross section of the I beam.

- DBDOUBLE HEAD_THICKNESS: this is the thickness of the top of the cross section of the beam.

Methods:

- *void set_external_height(double)*: this method allows the user to set the external height of the cross section of the beam.

- *void set_internal_height(double)*: this method allows the user to set the internal height of the cross section.

- *void set_top_width(double)*: this method allows the user to set the top width of the cross section, which is the width of the top part of the I beam.

- *void set_bottom_width(double)*: it allows the user to set the width of the bottom part of the cross section.

- *void set_thickness(double)*: it allows the user to set the thickness of the cross section.

4.4.13 class L_beam

Appendix C contains a drawing with all the dimensions of the L beam.

Attributes:

- DBDOUBLE HEIGHT: this is the total height of the cross section.

- DBDOUBLE WIDTH: it represents the total width of the cross section.

- DBDOUBLE THICKNESS: this attribute stores the value for the thickness of the L beam.

Methods:

- *void set_height(double)*: this method allows the user to set the total height of the cross section.
- *void set_width(double)*: this method allows the user to set the total width of the cross section, which is the width of the top part of the T beam.
- *void set_thickness(double)*: this method allows the user to set the thickness.

4.4.14 class Curved_line

This class is subclassed directly from the top level class **Engineering_object**. Its attributes and methods are:

Attributes:

- DBDOUBLE LENGTH: this is the length of the chord of the arc.
- DBDOUBLE RADIUS: this attribute represents the radius of the arc.
- DBDOUBLE CHORD_HEIGHT: this attribute represents the height of the arc from the chord.
- DBDOUBLE ANGLE: this attribute represents the angle of the arc.
- DBDOUBLE NUMBER_OF_DIVISIONS: the number of divisions represents the number of the straight segments that the curved line will be constructed of.

Methods:

- *void set_radius(double)*: this method allows the user to set the value for the radius of the curved line.
- *void set_angle(double)*: this method is for setting the angle of the curved line.
- *void set_number_of_divisions(int)*: this method allows the user to set the number of divisions that the curved line will be created from.

- *void set_length(double)*: it sets the length of the arc.
- *void set_chord_height(double)*: it sets the chord height.

4.4.15 class Surface

This class is subclassed directly from the top level class. It is the parent class of all the planar surfaces and plates.

Attributes:

- DBDOUBLE X_DIMENSION: it stores the value of the x dimension of the plate.
- DBDOUBLE Y_DIMENSION: it stores the value of the y dimension of the plate.
- DBINT PLANE: this attribute stores the plane that the surface will be created.

Methods:

- *void set_x_dimension(double)*: it allows the user to set the x dimension of the surface.
- *void set_y_dimension(double)*: it allows the designer to set the y dimension of the surface.
- *void set_plane(int)*: this method allows the user to set the plane that the surface will be created.

4.4.16 class Surface_rect_plate

Attributes:

- DBDOUBLE Z_DIMENSION: this attribute stores the z dimension of the surface.

Methods:

- *void set_z_dimension(double)*: this method allows the user to set the z dimension of the surface.

4.4.17 class Surface_circ_plate

Attributes:

- DBDOUBLE RADIUS: it stores the radius of the circular plate. This plate is constructed as a cylinder in GNOMES, so this is the radius of the cylinder.
- DBDOUBLE HEIGHT: this is the height of the circular plate, i.e. the height of the cylinder.
- DBINT NUMBER_OF_DIVISIONS: this attribute is the number of divisions that the cylinder will be constructed of.

Methods:

- *void set_radius(double)*: this method allows the user to set the radius of the circular plate.
- *void set_height(double)*: this method allows the user to set the height of the circular plate.
- *void set_number_of_divisions(int)*: it sets the number of divisions of the circular plate.

4.4.18 class Surface_rect_plate_solid

This class is a child class of Surface_rect_plate. It has no additional attributes, but its purpose is to create a solid rectangular plate.

4.4.19 class Surface_rect_plate_hollow

This class is a child class of Surface_rect_plate. The only additional attribute is the thickness of the cross section.

Attributes:

- DBDOUBLE THICKNESS: this attribute stores the thickness of the hollow cross section.

Methods:

- *void set_thickness(double)*: this method sets the thickness of the hollow cross section.

4.4.20 class Surface_circ_plate_solid

This class is a subclass of Surface_circ_plate and its purpose is to create a solid plate. It has no additional attributes than its parent class.

4.4.21 class Surface_circ_plate_hollow

Attributes:

- DBDOUBLE THICKNESS: this attribute specifies the thickness of the hollow cross section of the plate.

Methods:

- *void set_thickness(double)*: this method allows the designer to set the thickness of the hollow cross section of the plate.

4.4.22 class Curved_surface

This is a subclass of the top level class. It is the parent class of all the curved surfaces and shells. It represents a single curved surface with zero thickness.

Attributes:

- DBDOUBLE LENGTH: this attribute is the length of the arc of the single curved surface.
- DBDOUBLE CHORD_HEIGHT: it stores the value of the height of the curved surface from the chord.
- DBDOUBLE WIDTH: this is the width of the curved surface.
- DBDOUBLE RADIUS: this is the radius of the curved surface.

- DBDOUBLE ANGLE: it is the angle that the curved side of the surface has.
- DBINT NUMBER_OF_DIVISIONS: this attribute specifies the number of divisions that the curved surface will be constructed of.

Methods:

- *void set_length(double)*: it sets the length of the curved surface.
- *void set_chord_height(double)*: it sets the maximum height of the surface from the chord.
- *void set_width(double)*: it sets the width.
- *void set_radius(double)*: it assigns a value for the radius.
- *void set_angle(double)*: it assigns a value for the angle of the curved surface.
- *void set_number_of_divisions(int)*: it sets the number of divisions.

4.4.23 class Single_curved_surface

This class is subclassed from the Curved_surface class. The difference between the two classes is that the current one has thickness.

Attributes:

- DBDOUBLE THICKNESS: this is the thickness of the curved surface.

Methods:

- *void set_thickness(double)*: this method sets the thickness.

4.4.24 class Double_curved_surface

Attributes:

- DBDOUBLE THICKNESS: it represents the thickness of the double curved surface.

Methods:

- *void set_thickness(double)*: this method sets the thickness.

4.4.25 class Engineering_assembly

Appendix C contains drawings with the dimensions of the Engineering assemblies.

Attributes:

- DBINT NUMBER_OF_BAYS: this is the attribute that stores the value for the number of bays that the engineering assembly consists of.
- DBINT NUMBER_OF_STORIES: this attributes specifies the number of stories that the assembly consists of.
- DBDOUBLE ASSEMBLY_HEIGHT: this is the height of the assembly.

Methods:

- *void set_number_of_bays(int)*: it allows the user to specify the number of bays.
- *void set_number_of_stories(int)*: it allows the user to specify the number of stories that the assembly will have.
- *void set_assembly_height(double)*: this method sets the height of the assembly.

4.4.26 class Truss

This class has no additional attributes than its parent class, the **Engineering_assembly**. The difference between the two classes is that the Engineering_assembly may be a truss or a frame. If it is a truss, then it is single-story. The frame might be multi-story. So for a truss the number of stories is one. The class **Engineering_assembly** displays only the outline of the object.

4.4.27 class Frame

This class has no additional attributes than its parent class, the **Engineering_assembly**. The frame can be multi-story. The **Engineering_assembly** displays only the outline of the object.

4.4.28 class Cuboid

Attributes:

- DBDOUBLE LENGTH: it represents the length of the cuboid.
- DBDOUBLE HEIGHT: this is the height of the cuboid.
- DBDOUBLE WIDTH: this attribute stores the width of the cuboid.

Methods:

- *void set_length(double)*: this method allows the user to set the length of the cuboid.
- *void set_height(double)*: this method allows the user to set the height of the cuboid.
- *void set_width(double)*: it sets the width of the cuboid.

4.4.29 class Cone

Attributes:

- DBDOUBLE RADIUS: this is the radius of the base of the cone.
- DBDOUBLE HEIGHT: it represents the height of the cone.
- DBINT NUMBER_OF_DIVISIONS: this attribute specifies the number of divisions that the cone will be created from.

Methods:

- *void set_radius(double)*: this method sets the radius of the cone.
- *void set_height(double)*: it sets the height.
- *void set_number_of_divisions(int)*: it sets the number of divisions.

4.4.30 class Cylinder

Attributes:

- DBDOUBLE RADIUS: this is the radius of the cylinder.
- DBDOUBLE HEIGHT: this attribute stores the value for the height of the cylinder.
- DBINT NUMBER_OF_DIVISIONS: this attribute stores the value for the number of divisions of the cylinder.

Methods:

- *void set_radius(double)*: this method allows the user to set the radius of the cylinder.
- *void set_height(double)*: it allows the user to set the height of the cylinder.
- *void set_number_of_divisions(int)*: this method sets the number of divisions of the cylinder.

4.4.31 class Circle

Attributes:

- DBDOUBLE RADIUS: this attribute specifies the radius of the circle.
- DBINT PLANE: this is the plane that the circle will be created on.

Methods:

- *void set_radius(double)*: this method allows the user to set the radius of the circle.
- *void set_plane(int)*: this method allows the user to set the plane of the circle.

4.4.32 class Rectangle

Attributes:

- DBDOUBLE LENGTH: this attribute is the length of the rectangle.
- DBDOUBLE WIDTH: it represents the width of the rectangle.
- DBINT PLANE: it stores the value for the plane that the rectangle will be created on.

Methods:

- *void set_length(double)*: this method allows the user to set the length of the rectangle.
- *void set_width(double)*: this method allows the uset to set the width of the rectangle.
- *void set_plane(int)*: this method sets the plane of the rectangle.

4.4.33 class Sphere

Attributes:

- DBDOUBLE RADIUS: it stores the value for the radius of the sphere.
- DBINT NUMBER_OF_DIVISIONS: this attributes stores the number of divisions that the shpere will be constructed of.

Methods:

- *void set_radius(double)*: this method allows the user to store the value for the radius of the sphere.
- *void set_number_of_divisions(int)*: this method allows the designer to set the number of divisions that the sphere will be constructed from.

4.4.34 class Parametrized_object

This class is a friend class to the class Engineering_object.

Attributes:

- DBSTRUCT: this is a structure that contains the following attributes:
  - DBINT TYPE: this is the type of the object, i.e. it might be a cuboid, or a cylinder, or a T beam etc.
  - GNVECTOR ROT: this vector stores the total rotation that this object has with respect to the beginning of the axis.
  - GNVECTOR TRANS: this vector stores the total translation of this object with respect to the beginning of the axis.
  - DBINT PARAMETERS[8]: this attribute is an integer array of size 8. The integers stored are either 1 or 0. The value of 0 corresponds to a parameter dimension, while the value of 1 corresponds to a constant dimension. For example, if the value of "parameters[0]" is 1, this means that the first dimension of the specified object is constant, otherwise if the value is 0, the first dimension of the specified object is variable and the user will be asked to give a value for it, when he/she creates the parametrized object.
  - DBDOUBLE VALUES[8]: this attribute is an array of size 8, that stores "double" numbers. The numbers stored are the values of all the constant dimensions. If a dimension is constant, then when the object is being parametrized, the value of this constant is stored at the same position where the array "parameters" has the value of 1 for this constant. If the dimension is variable, then this array stores the value of 0.
  - ENGINEERING_OBJECT *COMPL: this attribute is for the case that the object to be parametrized contains a difference or intersection of other objects. In this case this difference or intersection remain as they are, without any change on them.

- PAR_CON OBJECTS[100]: this attribute is an array of structures of size 100 and it stores all the objects that the assembly to be parametrized, contains. The total number of objects that the assembly can contain is 100.

- DBCHAR PARAMETRIZED_NAME[50]: this array stores the name of the object being parametrized.

- DBINT NUMBER_OF_OBJECTS: this attribute contains the total number of objects that the assembly consists of.

Methods:

- *Parametrized_object(Engineering_object*,char*)*: this constructor is used for the parametrization of an object. The first argument is the address of the object that will be parametrized, while the second argument is the name that the parametrized object will take.

- *Parametrized_object()*: this is the destructor of the class.

- *void set_all_parameters()*: this method is used when the designer wants to create an object from an already parametrized one. Then this method asks for the values of the variables.

- *void set_one_parameter(int,int,int,double)*: this method is used when the user wants to create an object from an already parametrized one, but he/she does not want to set all the parameters, but just one. This method sets the value for one parameter only.

- *void find_parametrized_from_name(char *)*: this method takes as argument the name of a parametrized object and finds its address. Then this address is passed to the method *create_obj_from_param*, which asks the user the dimensions and then creates an engineering object from the parametrized one.

- *create_obj_from_param(Parametrized_object*)*: this method creates an *Engineering_object* from a *Parametrized_object*. The address of the *Parametrized_object* is passed as argument to this method.

4.5 Examples

4.5.1 General Description

This section contains examples from the creation of various engineering objects, created through the GAB module. The display of these objects is performed through the GRAPHITI user interface. GRAPHITI was built as an interface for the GNOMES solid modeling engine and does not have any provisions for the GAB module. Nevertheless, the GAB objects have the same structure as the GRAPHITI primitives, because they are both created by GNOMES. Therefore the GAB objects can be displayed in the GRAPHITI user interface, provided that they are created by some other means. This can be achieved by creating the GAB objects through a computer program, that directly calls the GAB methods. Then these objects can be called, displayed and manipulated by the GRAPHITI user interface.

A computer program was written in the C++ language, to illustate the use of the GAB module. The program performs the direct creation of some engineering objects, for the case that all the information is known. In this computer program all the attribute values have been set by using the separate methods for each attribute. Alternatively, the method *set_attribute_values* could have been used, that asks the user for the attribute values. The user has to open the database first by choosing the appropriate option. Then he/she can create the desired object. When the object has been created, the database should be closed.

Printouts of some of the objects created by this program and displayed through the GRAPHITI user interface are included in Appendix D.

4.5.2 Evolving Shape Description of a Beam With Hollow Rectangular Cross Section

For this example it is assumed that a designer wants to create a beam. Initially the type of beam is not known. The only information available is the length of the space that the beam will be positioned. So the designer creates a straight line with the

appropriate length.

Then, as the design evolves, the user decides to create a rectangular beam. He/she acquires the outside dimensions for the width and height of the cross section, but no other information is available for example if the beam will be hollow or solid. The GAB module allows the gradual shape evolution of the straight line to a line with rectangular cross section.

At a later stage of the design, the user decides that the beam will be hollow rectangular, but no information is available about the thickness of it. The GAB module allows him/her to upgrade the line with rectangular cross section to a line with hollow rectangular cross section. Since the user does not give any value for the thickness of the hollow beam, the internal levels of abstraction will display an abstract representation of it, which is a hollow beam with zero thickness. So this abstract representation consists of 2D surfaces.

Then at a later stage the user calculates a value for the thickness of the beam. The GAB module allows him/her to assign the value for the thickness and redisplay the beam in full detail.

This example shows how the GAB module allows the evolving shape description through external and internal levels of abstraction. The code that creates this example is included in the following lines.

```
#include "Gab_object.h"
#include "Gab_persistent.h"
#include "GNmanager.h"

#include "gnomes.h"
#include "Straight_line.h"
#include <E/trans.h>

GNmanager * gnomes_manager ;
```

10

```
////////////////////////////////////////////////////
```

```
//THIS IS AN EXAMPLE THAT ALLOWS THE EVOLVING SHAPE  //
//DESCRIPTION OF A STRAIGHT LINE TO A BEAM WITH      //
//HOLLOW RECTANGULAR CROSS SECTION. ALSO THE INTERNAL//
//LEVELS OF ABSTRACTION ARE USED, WHEN THE PROGRAM   //
//ASKS THE CREATION OF THE HOLLOW RECTANGULAR CROSS  //
//SECTION, BUT WITHOUT HAVING SPECIFIED THE THICKNESS//
//OF THE BEAM.                                       //
////////////////////////////////////////////////////////                    20


main(){

  //INSTANTIATE THE GNOMES MANAGER
  GNmanager* amanager = new GNmanager("George");
  gnomes_manager =  amanager;

  //CREATE THE DATABASE
  E_BeginTransaction();                                                     30
  amanager->create_db("examples");


  //CREATE A STRAIGHT LINE AS A GAB_OBJECT
  Gab_object * strline = new (objectC) Gab_object();
  strline->create_object(Gab_object::STRAIGHTLINE,"strline");
  strline->object->set_length(310);//DEFINE THE LENGTH
  strline->object->set_axis(1);//DEFINE THE AXIS
  strline->object->display();//DISPLAY THE LINE
                                                                            40

  //UPGRADE THE STRAIGHT LINE TO A LINE WITH
  //RECTANGULAR CROSS SECTION
  strline->upgrade(Gab_object::LINERECTCROSSSECTION);
  strline->object->set_width(8);//SET THE WIDTH OF THE
                        //LINE WITH RECTANGULAR
                        //CROSS SECTION
  strline->object->set_height(7);//SET THE HEIGHT
```

```
strline->object->display();//DISPLAY THE OBJECT
                                                                    50

//UPGRADE THE LINE WITH RECTANGULAR CROSS SECTION
//TO A LINE WITH HOLLOW RECTANGULAR CROSS SECTION
strline->upgrade(Gab_object::LINERECTHOLLOW);

//HERE WE ASK TO DISPLAY THE OBJECT WITHOUT HAVING
//SET THE THICKNESS. THIS WILL CAUSE THE INTERNAL
//LEVELS OF ABSTRACTION TO DISPLAY AN ABSTRACT
//REPRESENTATION OF THE OBJECT
strline->object->display();                                         60


strline->object->set_thickness(2);//SET THE THICKNESS
strline->display();//DISPLAY THE OBJECT IN FULL DETAIL

//CLOSE THE DATABASE
amanager->close_db();
E_CommitTransaction();

                                                                    70
}
```

4.6 Chapter Summary

This chapter contains a detailed description of the GAB classes. Initially there is a generic description of the purpose and knowledge included in each class. This is followed by a detailed description of the attributes and methods. Many methods are virtual and they provide a polymorphic interface for response to messages. At the end there are examples that show the use of the GAB capabilities.

Chapter 5

GRAPHITI Graphical User Interface

5.1 Chapter Overview

GRAPHITI is the graphical user interface for the GNOMES solid modeling engine. It was built in order to replace the older user interface ELF, described in the GNOMES design document [29], and to provide increased functionality and the capability to work in a shared environment. Initially GRAPHITI was built on top of the PHIGS graphics library [3],[19], but then it was changed to the HOOPS graphics library [11]. The reasons for this change were that:

- our version of PHIGS did not support 3D viewing capabilities (z-buffering), while HOOPS includes a software z-buffer.
- the X-server needs the PEX extension to run the PHIGS library, while HOOPS can run on top of an ordinary X-server. This gives HOOPS the flexibility to run from a remote server that does not support the PEX extension.

GRAPHITI was built using the object oriented paradigm. The code is implemented in the C++ language. Also the menus and the windows were created using the object oriented approach of C++ and OSF/Motif, described in [32].

5.2 Motivation

ELF was the graphical user interface for the GNOMES solid modeling engine before the creation of GRAPHITI. It was a menu-driven interface based on OSF Motif and the HOOPS graphics library. The ELF user interface found to have some limitations, and it was decided that a new user interface should be built that would combine the object oriented paradigm, together with more user facilities.

5.3 Objectives

GRAPHITI was built in order to replace the ELF user interface and provide the user with the following features:

- the capability to work in a shared environment.
- a more consistent menu layout and functional grouping, in order to make the interface more intuitive [5].
- the capability for the direct creation and manipulation of objects by using the mouse [5].
- command and status lines. The status line provides feedback for the current state and also it provides information messages. The command line is a fully functional alternative to the mouse/menu system [5].

5.4 GRAPHITI Environment

The GRAPHITI user interface was built as a project for course "Computer Aided Engineering II: Software Engineering for CAE Systems" 1.552 of M.I.T. ([5] and [4]), and it was completed as part of this thesis.

Initially, the graphics part of GRAPHITI was built on top of the PHIGS graphics library. But then it became clear that PHIGS imposed some limitations to the use of the interface. The interface was built in order to work in a shared environment,

where everyone could access it and use it, even from remote servers. The limitations that PHIGS imposed are:

- our version of the PHIGS library did not support 3D viewing capabilities (z-buffering).
- the PHIGS library could run only on workstations with the PEX extension.
- PHIGS is a low level graphics library and the code for the graphics part of the interface was very long.

For the above mentioned reasons it was decided that PHIGS is not the best library for our purpose of a shared environment and it was decided to change it back to the HOOPS graphics library. HOOPS has the following advantages over PHIGS for a shared environment:

- it is built on top of the X windows system, so it does not require any extension to the X-server. This gives the ability to run it from any remote server that supports the X-windows system.
- it has z-buffering capabilities built in it.
- it is a higher level graphics library than PHIGS, and therefore is easier to use and shorter in code length.

The environment of GRAPHITI is supported by and developed on the platforms of:

- *Language*: it is based on an object-oriented approach and it is coded in the C++ language.
- *Menus*: the creation of the menus is based on the object-oriented paradigm of C++ and OSF/Motif, and it uses the principles discussed in reference [32].
- *Operating System*: GRAPHITI is intended for the UNIX operating system, although the use of standard library calls and the MIT X11 Windowing System, enhances its ability to be ported to other compatible operating environments.

- *Hardware*: the development environment is the Sun system installed in the Intelligent Engineering Systems Laboratory (IESL) of the Civil and Environmental Department of M.I.T.

5.5 Background

All Graphiti windows and menus are derived from a base set of C++ classes, that form a simple application framework. This framework is called Motifapp and it supports features found in most Motif applications. It is described by Young in his book [32]. The idea behind this framework is to make Motif an object oriented environment and to create classes of objects that are commonly used in all Motif applications. These classes support the Motif programmer by providing window and menu templates that can be created by a simple instantiation of a class.

This framework contains five major classes:

1. **Application class**
2. **Main Window and Menu Window class**
3. **Cmd class**
4. **MenuBar class**
5. **Dialog Manager class**

The **Application class** handles the initialization and the events of the X window application. The **MainWindow class** provides the main window for an application and acts as a manager of all the other windows. The **Cmd class** is an abstract class, that encapsulates the behavior of a command object. Also it has the capability of enabling and disabling lists of commands and also supports a one level "undo" facility. The **MenuBar class** creates a window that contains a menubar at the top and the client application is required only to create list of commands and link them to the appropriate buttons of the menubar. The **Dialog Manager class** has

predefined pop-up dialogs, and the user can display his/her own messages on these pop-up windows. The classes **Application, Main Window, MenuBar, Dialog Manager** are subclassed from the higher level **UIComponent** class of the Motifapp framework.

5.6 General Description of GRAPHITI

A generic hierarchical diagram of the GRAPHITI structure is presented in figure 5-1.

The top level window contains all the other windows. The hierarchy is as follows (directly taken from [4]):

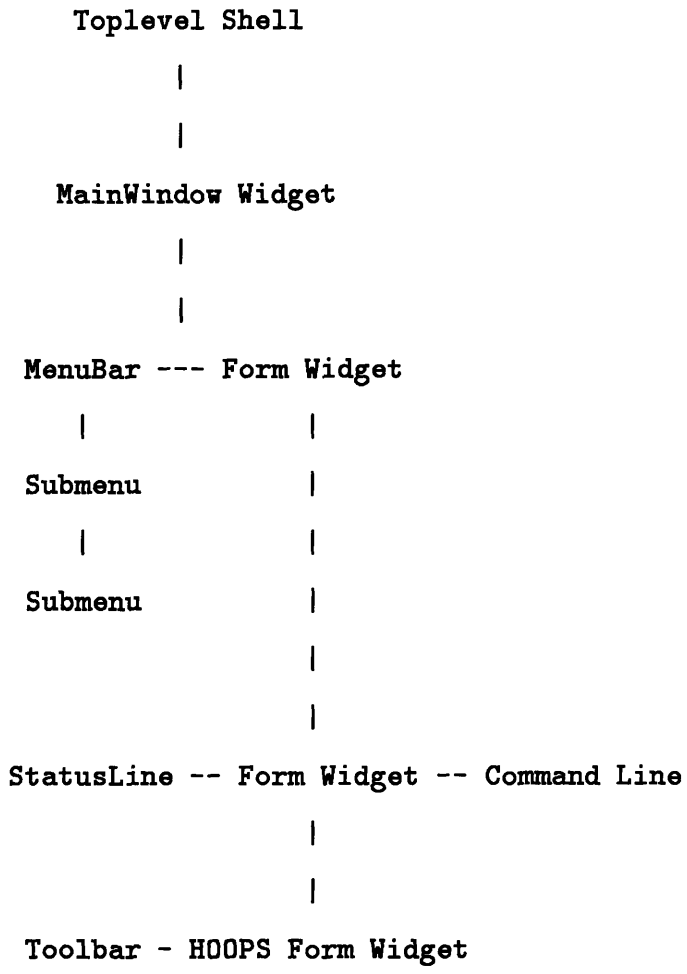

Figure 5-2 represents the GRAPHITI main state diagram.

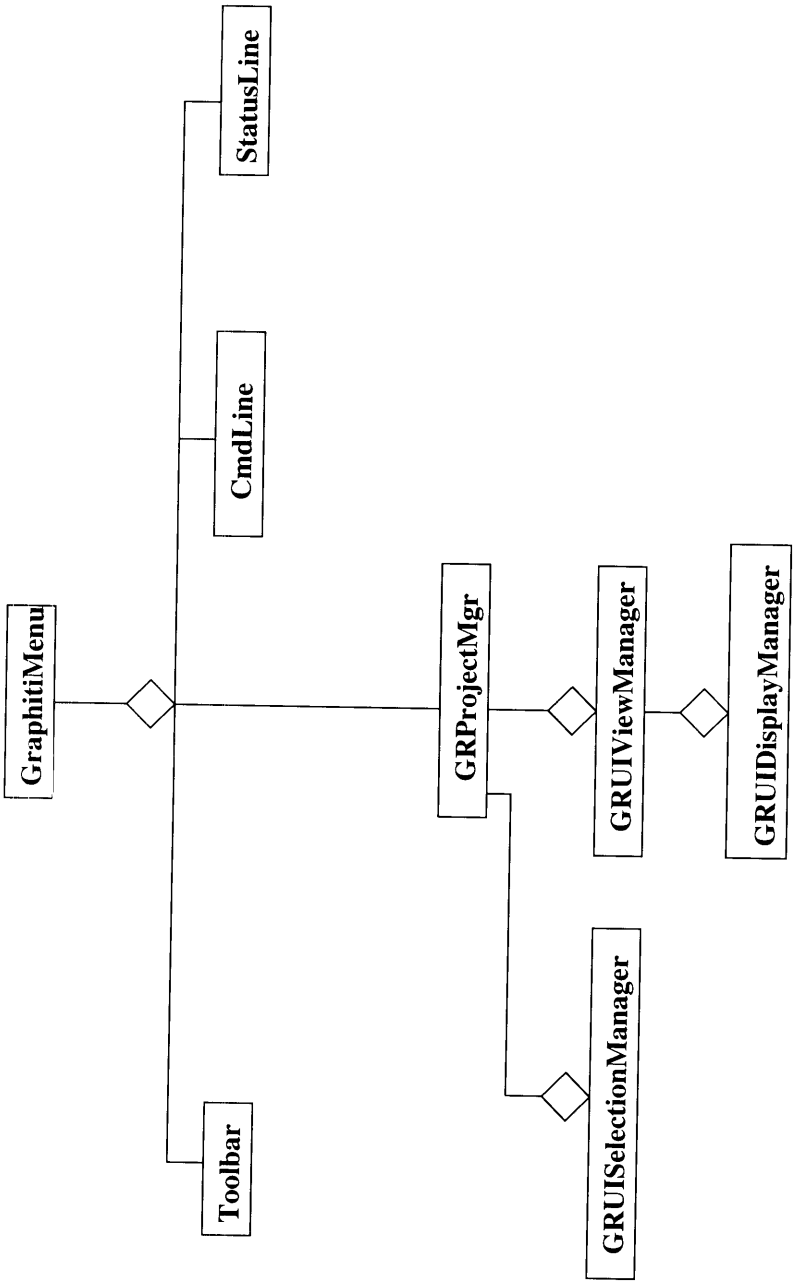

Figure 5-1: GRAPHITI Class Object Diagram

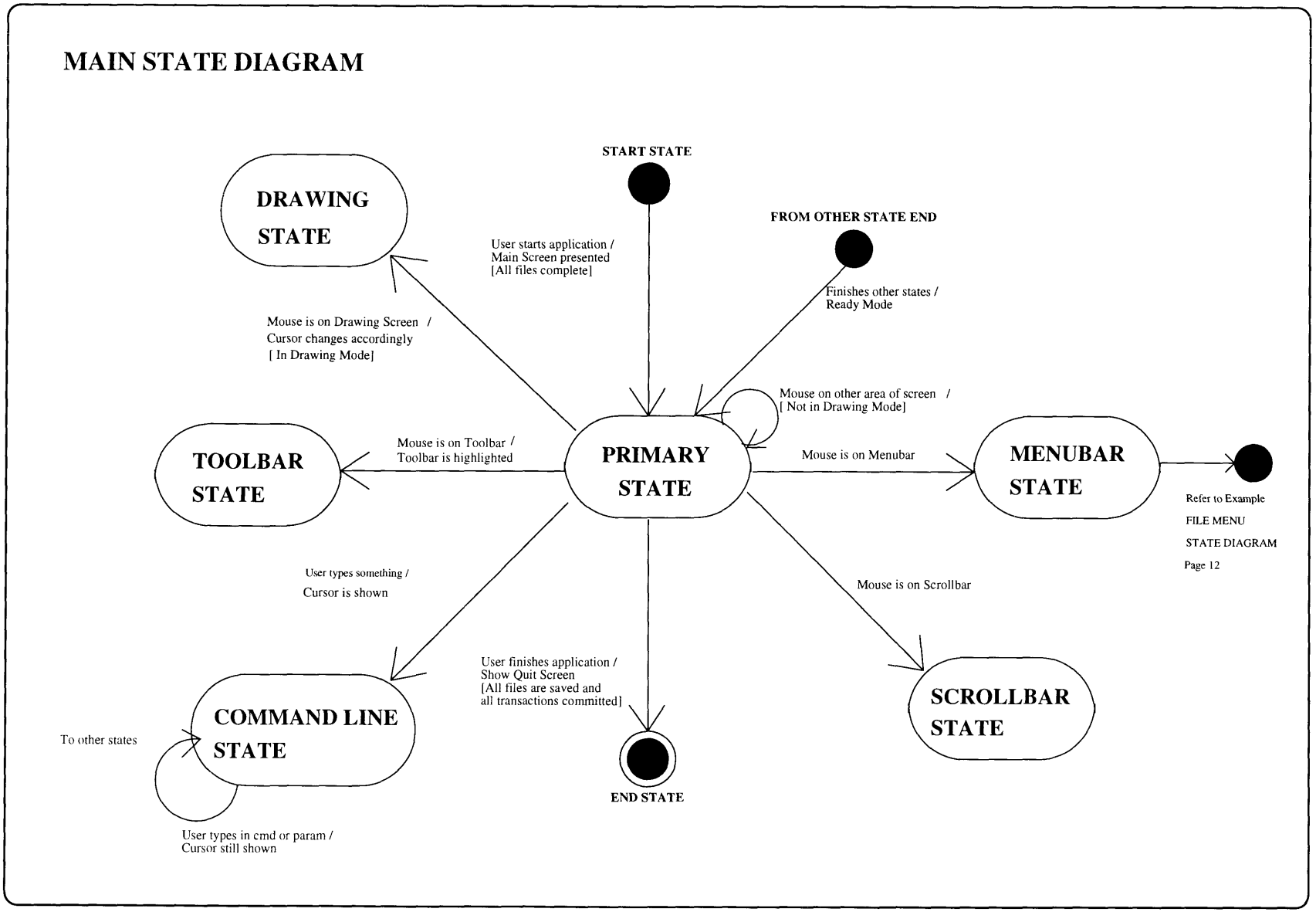

Figure 5-2: GRAPHITI Main State Diagram

Many of the GRAPHITI classes were adapted from the Motifapp framework in order to provide the facilities required for this interactive user interface. The classes that create the main window, the widgets and the menus are derived from the Motifapp framework. So, the **GraphitiMain** class is subclassed from **MenuWindow** of the Motifapp. Also the MainWindow widget is an instantiation of the Motifapp class **MenuWindow**. This object is the top level widget in the GRAPHITI module and it manages the entire environment, by setting up instances of the lower level classes. The **MenuWindow** constructor performs the X Window initialization together with the initialization of GNOMES and the database.

A brief description of the important classes follows:

- **GraphitiMain**: this class initializes the X windows and the other required libraries and performs the event handling.
- **GraphitiMenu**: it creates the main window and all the menus.
- **ObjList**: it creates a list that holds the objects of the current database.
- **GRProjectMgr**: this class is the "project manager" and it acts as an interface between the user interface and the GNOMES solid modeling engine. It provides the methods for the Boolean operations and all the other facilities needed for the interaction between the interface and the solid modeler.
- **GRUIViewManager**: it is responsible for displaying the objects on the screen. It takes the address of a GNOMES object from the **GRProjectMgr** and creates a HOOPS object. Also it contains all the methods for changing the viewpoint of the camera and the way object appear on the screen.
- **GRUIDisplayManager**: this class takes a HOOPS object from the previous class and displays it on the three different windows in perspective, top and right view.
- **GRUISelectionManager**: it contains all the facilities required for the selection of a displayed object either by mouse or from a list. The proper function-

ality of this class is very important, because all the operations in GRAPHITI require some objects to be selected.

- **PromptDialogManager**: it is an adaptation of the **DialogManager** class described in [32] and it creates a pop-up panel of sliders and control buttons.
- **SliderDialogManager**: it is an adaptation of the **DialogManager** of the Motifapp framework that creates a pop-up window of sliders and control panels, different than that of the previous class.
- **StatusLine**: this class creates and handles the status line.
- **Toolbar**: it creates and handles the tool bar.
- **TbCmdLine**: it contains all the methods required for the creation of a GNOMES line through the **GRProjectMgr** class.
- **TbCmdSquare**: it contains all the calls required for the creation of a GNOMES square through the **GRProjectMgr** class.
- **TbCmdArc**: this class is responsible for the creation of an arc.
- **TbCmdCone**: this class creates a cone through the class **GRProjectMgr**.
- **TbCmdCylinder**: it creates a cylinder
- **TbCmdSphere**: it contains all the methods that allow the user to create a GNOMES sphere.
- **TbCmdCube**: it allows the user to create a cube.
- **TbCmdCircle**: it allows the user to create a circle.
- **TbCmdPolygon**: it allows the user to create a Polygon.
- **AttrDialogManager**: it is an adaptation of the **DialogManager** class described in citeC++-Motif that creates a pop-up window of text fields and reads the strings that the user specifies on these fields.

- **ChooseColorCmd**: this class creates a pop up window that allows the user to choose a color. Then this class handles the change of the color of the selected objects. The pop up window for selecting a color is implemented in the Motifapp framework.

- **CmdLine**: this class is responsible for the command line creation and handling.

- **CmdRich**: this is a base class for all command objects.

Detailed descriptions of the above classes are given in [4].

5.7 Chapter Summary

GRAPHITI is the graphical user interface of the GNOMES geometric engine. GRAPHITI was built to replace the older interface ELF, which imposed limitations to the user. It is built using the object oriented paradigm and it is coded in the C++ language. The creation of the windows and menus is based on the principles of C++ and OSF/Motif discussed in reference [32]. The graphics library used is HOOPS, which provides for the use of GRAPHITI in a shared environment, where everyone can access it from remote workstations. The only requirement is that the remote workstations should support an X-server.

Chapter 6

Summary and Future Work

6.1 Summary

The design of engineering objects is a complicated process and all the information is not known from the initial stages. Also the designers want to associate some properties or features with each object other than geometric representations. In addition to that, designers want to have the flexibility of creating engineering objects fast and be able to modify them easily. Since each area of engineering has different objects associated with it, it is very useful to create different computer modules for each area, that include the objects most widely used in this particular area.

In order to support the conceptual design of engineering objects, the computer module GAB (Geometric Abstractions) was developed in C++ language, on top of the GNOMES solid modeler. The module provides the capability of creating a set of engineering objects widely used in Ocean and Civil Engineering like beams, plates, shells etc. Also it allows evolving shape design, through different levels of abstraction. This means that the user can display the objects from the initial stages of design, before all the information is known about an object. This can be achieved through the different levels of abstraction, which have been implemented as C++ classes in a hierarhical manner. The hierarchical tree of the classes has been designed based on the shape classification described in [23]. At the top level there is the base class which contains all the utilities for the classes. Also it allows the user to assign to each object

features like bounding box, type of material, color, specific weight. At the next level there are the classes for the straight line, curved line, surface, and curved surface and also all the primitive objects like cuboid, cylinder etc. From the straight line all the beams and elongated objects are derived by applying to it different cross sections. From the surface all the plates and surface-like objects are derived, while from the curved surface, all the three dimensional curved objects and shells are derived.

The GAB modules has two different types of abstraction. The first one is external abstractions between the classes. If the designer does not know the type of object, then he/she can start from the top level classes and when the knowledge about the object is acquired, proceed through the class hierarchy. Then there is also the internal type of abstractions. These are performed internally in each class. If the exact type of object is known from the initial stages of design, but not all the information about it has been obtained, then an abstraction of this object is represented on the screen. Each class has the knowledge to display the object, based on the information available. If some information is missing, then the best possible object is displayed. The knowledge for performing the representation of the objects through internal abstractions is rule-based and it is contained inside the method that displays the object for each class.

Also the GAB module provides the capability of creating user defined objects. This is done through the parametrization of an object. For the first time, the user has to create the engineering object using the GAB capabilities and the facilities provided by the user interface for the GNOMES solid modeling engine. Then the module asks the user to specify which dimensions will be parameters and also asks about a name. The next time that this name will be called, the user will be asked only for the values of the parameters and the object will be created at the position of the initial object, but with the new values for the parameters as dimensions. This part of the module needs further development and the provision for non valid object checking.

6.2 Future Work

The GAB module can be developed further by the addition of more engineering objects. For example the sweeping facilities of GNOMES could be used for the creation of curved lines with a particular cross section that the user would define.

Also the class **Parametrized_object** for parametrizing an object does not check for inconsistent objects. The parametrized objects change their parameter-dimensions with respect to the center of volume. A future improvement would allow the user to specify a point on the object with respect to which the parameter-dimensions would change. Also the user should be allowed to "glue" the objects, that constitute the assembly, at a particular point, so that if one parameter-dimension of an object changes, the other "glued" object will have to move with the changing dimension.

Also a future improvement of the GAB module would provide for the creation of perpendicular edges to the surface for the classes **Single_curved_surface** and **Double_curved_surface**, as described in Section 4.2.

The GRAPHITI user interface was built before the creation of the GAB module. Therefore, it was built to support the creation of GNOMES objects. Nevertheless the objects created by GAB can be displayed in GRAPHITI, because they are built as GNOMES objects, provided that these objects are created by a computer program. A future improvement of the GRAPHITI user interface, would support the direct creation of GAB objects through the interface.

Appendix A

APPENDIX A: GNOMES HIERARCHY

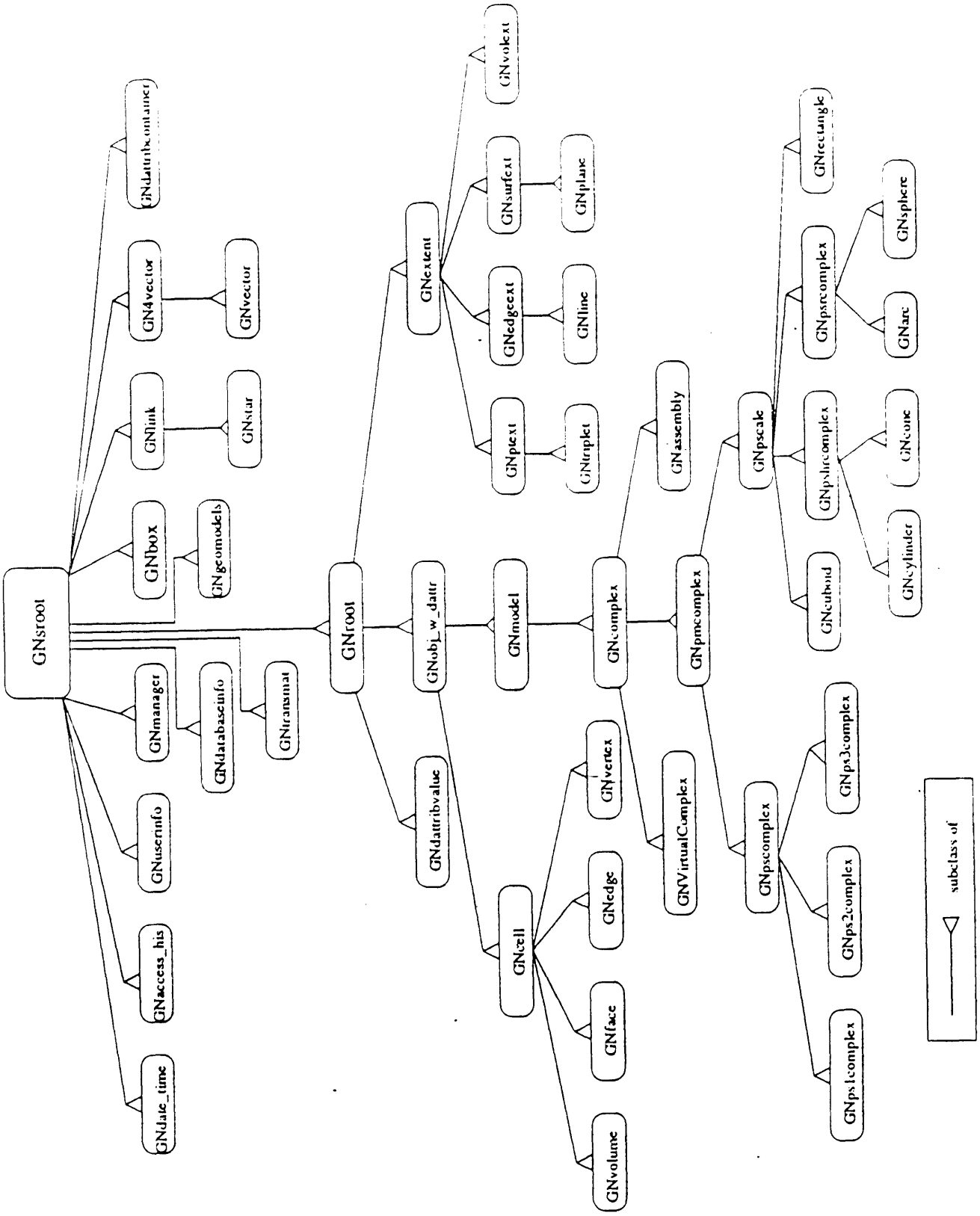

Figure A-1: Class Hierarchy of GNOMES

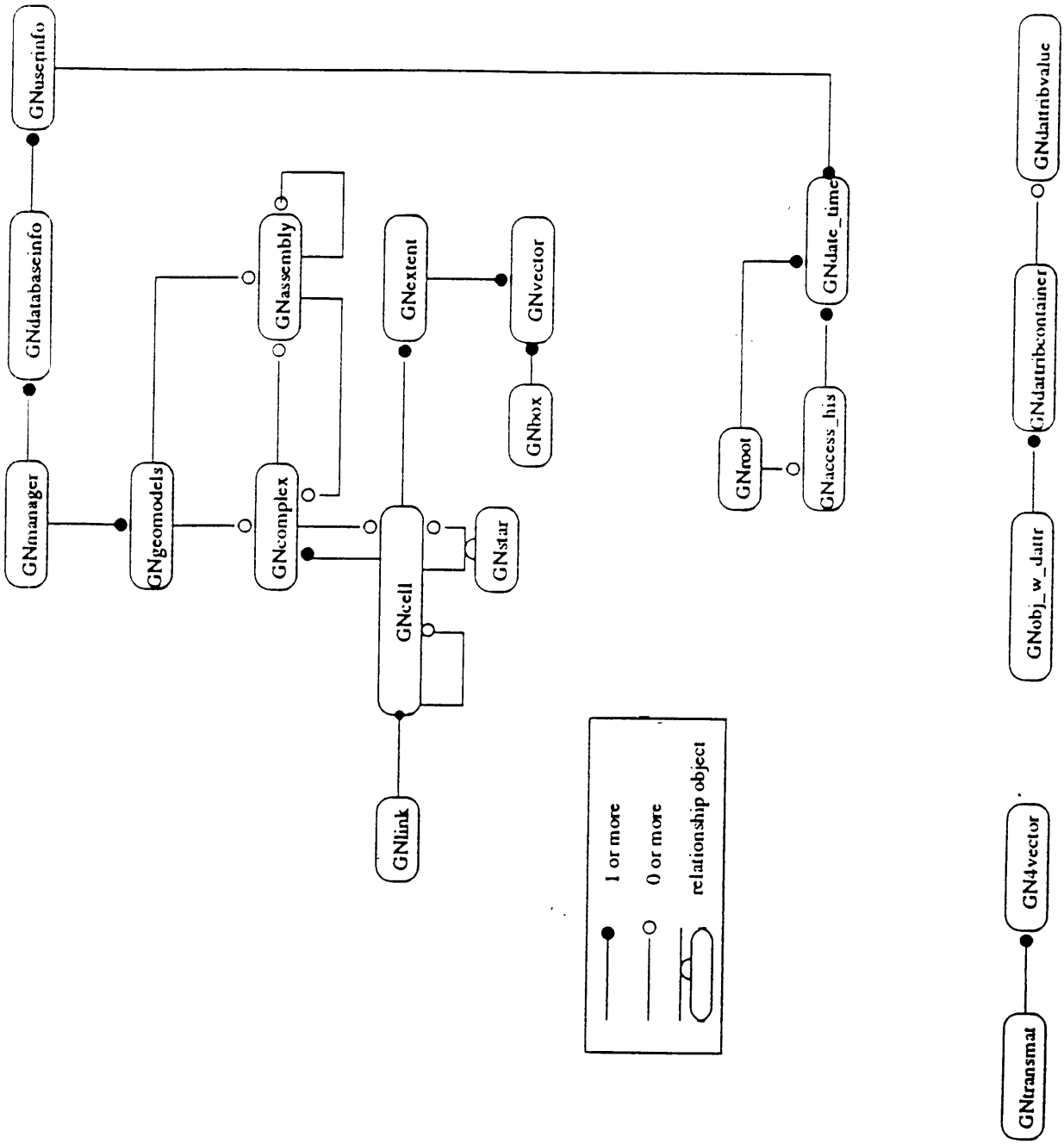

Figure A-2: Component Hierarchy of GNOMES

Appendix B

APPENDIX B: GAB HEADER FILES

```
#ifndef COMPLEXES
#define COMPLEXES

#include <iostream.h>
#include "Engineering_object.h"

//THIS FILE BUILDS OBJECT ABSTRACTIONS AROUND
//SOME OF THE GNOMES CSG PRIMITIVES.
//IN THIS FILE WE DO NOT HAVE CLASSES FOR ARC
//AND LINE BECAUSE THEY ARE CREATED DIRECTLY                    10
//FROM THE CLASSES STRAIGHT_LINE AND CURVED_LINE

//IN EACH OF THE FOLLOWING: THE UPGRADE CONSTRUCTOR
//IS USED TO UPGRADE THE OBJECT FROM AN EXISTING
//OBJECT (COPY ALL ATTRIBUTES OF OBJECT).
//THIS CONSTRUCTOR DELETES THE OLD OBJECT !!

//////////////CLASS CUBOID ////////////////////////
dbclass Cuboid : public Engineering_object
{                                                               20
public:
```

```
  Cuboid(char *);
  Cuboid(Engineering_object *); //UPGRADE CONSTRUCTOR
  Cuboid(Cuboid *);//COPY CONSTRUCTOR
  Cuboid(char*,double,double,double,GNvector,GNvector,double);
                    //CONSTRUCTOR FOR
                    //DIRECT CREATION OF AN
                    //OBJECT BY GIVING ITS
                    //ATTRIBUTES
  virtual ~Cuboid();                                                    30
  virtual void display();
  virtual void set_attribute_values();
  virtual void set_length(double);
  virtual void set_height(double);
  virtual void set_width(double);

protected:
  dbdouble length;
  dbdouble height;
  dbdouble width;                                                       40
  virtual void propagate_attribute_values();
  virtual int check_attribute_values();
};

////////////////CLASS CONE//////////////////////////

dbclass Cone : public Engineering_object
{
public:
  Cone(char *);                                                         50
  Cone(Engineering_object *); //UPGRADE CONSTRUCTOR
  Cone(Cone *); //COPY CONSTRUCTOR
  Cone(char*,double,double,int,GNvector,GNvector,double);
                    //CONSTRUCTOR FOR
                    //DIRECT CREATION OF AN
                    //OBJECT BY GIVING ITS
                    //ATTRIBUTES
```

```
  virtual ~Cone();
  virtual void display();
  virtual void set_attribute_values();                                    60
  virtual void set_radius(double);
  virtual void set_height(double);
  virtual void set_number_of_divisions(int);

protected:
  dbdouble radius;
  dbdouble height;
  dbint number_of_divisions;
  virtual void propagate_attribute_values();
  virtual int check_attribute_values();                                   70
};

//////////////////CLASS CYLINDER/////////////////////

dbclass Cylinder : public Engineering_object
{
public:
  Cylinder(char *);
  Cylinder(Engineering_object *); //UPGRADE CONSTRUCTOR
  Cylinder(Cylinder *); //COPY CONSTRUCTOR                                 80
  Cylinder(char*,double,double,int,GNvector,GNvector, double);
                    //CONSTRUCTOR FOR
                    //DIRECT CREATION OF AN
                    //OBJECT BY GIVING ITS
                    //ATTRIBUTES
  virtual ~Cylinder();
  virtual void display();
  virtual void set_attribute_values();
  virtual void set_radius(double);
  virtual void set_height(double);                                        90
  virtual void set_number_of_divisions(int);

protected:
```

```
  dbdouble radius;
  dbdouble height;
  dbint number_of_divisions;
  virtual void propagate_attribute_values();
  virtual int check_attribute_values();
};
                                                                    100
////////////////CLASS CIRCLE/////////////////////

dbclass Circle : public Engineering_object
{
public:
  Circle(char *);
  Circle(Engineering_object *); //UPGRADE CONSTRUCTOR
  Circle(Circle *); //COPY CONSTRUCTOR
  Circle(char*,double,int,int,GNvector,GNvector,double);
                    //CONSTRUCTOR FOR                               110
                    //DIRECT CREATION OF AN
                    //OBJECT BY GIVING ITS
                    //ATTRIBUTES
  virtual ~Circle();
  virtual void display();
  virtual void set_attribute_values();
  virtual void set_radius(double);
  virtual void set_plane(int);
  virtual void set_number_of_divisions(int);
                                                                    120
protected:
  dbdouble radius;
  dbint plane;
  dbint number_of_divisions;
  virtual void propagate_attribute_values();
  virtual int check_attribute_values();
};

//////////////CLASS RECTANGLE/////////////////
```

```
                                                                    130
dbclass Rectangle : public Engineering_object
{
public:
  Rectangle(char *);
  Rectangle(Engineering_object *); //UPGRADE CONSTRUCTOR
  Rectangle(Rectangle *); //COPY CONSTRUCTOR
  Rectangle(char*,double,double,int,GNvector,GNvector,double);
                    //CONSTRUCTOR FOR
                    //DIRECT CREATION OF AN
                    //OBJECT BY GIVING ITS                          140
                    //ATTRIBUTES
  virtual ~Rectangle();
  virtual void display();
  virtual void set_attribute_values();
  virtual void set_width(double);
  virtual void set_length(double);
  virtual void set_plane(int);

protected:
  dbdouble length;                                                  150
  dbdouble width;
  dbint plane;
  virtual void propagate_attribute_values();
  virtual int check_attribute_values();
};

//////////////CLASS SPHERE/////////////////////////

dbclass Sphere : public Engineering_object
{                                                                   160
public:
  Sphere(char *);
  Sphere(Engineering_object *); //UPGRADE CONSTRUCTOR
  Sphere(Sphere *); //COPY CONSTRUCTOR
  Sphere(char*,double,int,GNvector,GNvector,double);
```

```
                    //CONSTRUCTOR FOR
                    //DIRECT CREATION OF AN
                    //OBJECT BY GIVING ITS
                    //ATTRIBUTES
  virtual ~Sphere();                                                    170
  virtual void display();
  virtual void set_attribute_values();
  virtual void set_radius(double);
  virtual void set_number_of_divisions(int);

protected:
  dbdouble radius;
  dbint number_of_divisions;
  virtual void propagate_attribute_values();
  virtual int check_attribute_values();                                 180
};

#endif COMPLEXES

                                                                        190
```

```
#ifndef CURVED_LINE
#define CURVED_LINE

#include <iostream.h>
#include "Engineering_object.h"

////////////////CLASS CURVED LINE/////////////
```

```
dbclass Curved_line : public Engineering_object
{                                                                    10
public:
  Curved_line(char *);
  Curved_line(Engineering_object *); //UPGRADE CONSTRUCTOR
  Curved_line(Curved_line*); //COPY CONSTRUCTOR
  Curved_line(char*,double,double,int,GNvector,GNvector,double);
                    //CONSTRUCTOR FOR
                    //DIRECT CREATION OF AN
                    //OBJECT BY GIVING ITS
                    //ATTRIBUTES
  virtual ~Curved_line();                                            20
  virtual void display();
  virtual void set_attribute_values();
  virtual void set_radius(double);
  virtual void set_angle(double);
  virtual void set_number_of_divisions(int);
  virtual void set_length(double);
  virtual void set_chord_height(double);

protected:
  dbdouble length; //THE LENGTH OF THE CHORD                         30
  dbdouble radius;
  dbdouble chord_height; //THE HEIGHT OF THE
                    //ARC FROM THE CHORD
  dbdouble angle;//THE ANGLE IS IN RADIANS
  dbint number_of_divisions;
  virtual void propagate_attribute_values();
  virtual int check_attribute_values();
};

#endif CURVED_LINE                                                   40
```

```
                                                                    50
#ifndef CURVED_SURFACE
#define CURVED_SURFACE

#include <iostream.h>
#include "Engineering_object.h"

//////////////////CLASS CURVED_SURFACE////////////////

dbclass Curved_surface : public Engineering_object
{                                                                   10
public:
  Curved_surface(){}
  Curved_surface(char *);
  Curved_surface(Engineering_object *);
  Curved_surface(Curved_surface*);
  Curved_surface(char*,double,double,double,int,GNvector,GNvector,double);
                    //CONSTRUCTOR FOR
                    //DIRECT CREATION OF AN
                    //OBJECT BY GIVING ITS
                    //ATTRIBUTES                                    20
  virtual ~Curved_surface();
  virtual void display();
  virtual void set_attribute_values();
  virtual void set_length(double);
  virtual void set_width(double);
  virtual void set_radius(double);
  virtual void set_angle(double);
  virtual void set_number_of_divisions(int);
```

```
  virtual void set_chord_height(double);
                                                                    30
protected:
  dbdouble length;
  dbdouble chord_height;
  dbdouble width;
  dbdouble radius;
  dbdouble angle;
  dbint number_of_divisions;
  virtual void propagate_attribute_values();
  virtual int check_attribute_values();
};                                                                  40

//////////////CLASS SINGLE_CURVED_SURFACE/////////

dbclass Single_curved_surface : public Curved_surface
{
public:
  Single_curved_surface(char *);
  Single_curved_surface(Curved_surface *);
  Single_curved_surface(Single_curved_surface*);
  Single_curved_surface(char*,double,double,double,int,double,GNvector,GNvector,double); 50
                    //CONSTRUCTOR FOR
                    //DIRECT CREATION OF AN
                    //OBJECT BY GIVING ITS
                    //ATTRIBUTES
  virtual ~Single_curved_surface();
  virtual void display();
  virtual void set_attribute_values();
  virtual void set_thickness(double);

protected:                                                          60
  dbdouble thickness;
  virtual void propagate_attribute_values();
  virtual int check_attribute_values();
};
```

```
//////////////CLASS DOUBLE_CURVED_SURFACE/////////////

dbclass Double_curved_surface : public Curved_surface
{                                                                    70
public:
  Double_curved_surface(char *);
  Double_curved_surface(Curved_surface *);
  Double_curved_surface(Double_curved_surface *);
  Double_curved_surface(char*,double,double,int,double,GNvector,GNvector,double);
                        //CONSTRUCTOR FOR
                        //DIRECT CREATION OF AN
                        //OBJECT BY GIVING ITS
                        //ATTRIBUTES
  virtual ~Double_curved_surface();                                  80
  virtual void display();
  virtual void set_attribute_values();
  virtual void set_thickness(double);

protected:
  dbdouble thickness;
  virtual void propagate_attribute_values();
  virtual int check_attribute_values();
};
                                                                     90
////////////////////////////////////////////////

#endif CURVED_SURFACE
```

100

110

```
#ifndef ENGINEERING_ASSEMBLY
#define ENGINEERING_ASSEMBLY

#include <iostream.h>
#include "Straight_line.h"

//TTHE ASSEMBLIES ARE DRAWN IN PLANE 1 (XZ)

//////////////CLASS ENGINEERING_ASSEMBLY////////////
                                                                    10
dbclass Engineering_assembly : public Straight_line
{
public:
  Engineering_assembly(){}
  Engineering_assembly(char *);
  Engineering_assembly(Straight_line *); //UPGRADE CONSTRUCTOR
  Engineering_assembly( Engineering_assembly*); //COPY CONSTRUCTOR
  Engineering_assembly(char*,double,double,double,GNvector,GNvector,double);
                    //CONSTRUCTOR FOR
                    //DIRECT CREATION OF AN                         20
                    //OBJECT BY GIVING ITS
```

```
                    //ATTRIBUTES
  virtual ~Engineering_assembly();
  virtual void display();
  virtual void set_attribute_values();
  virtual void set_number_of_bays(int);
  virtual void set_number_of_stories(int);
  virtual void set_assembly_height(double);

protected:                                                          30
  dbdouble number_of_bays;
  dbdouble number_of_stories;
  dbdouble assembly_height;
  //LENGTH IS INHERITED FROM THE STRAIGHT LINE
  virtual void propagate_attribute_values();
  virtual int check_attribute_values();
};

////////////////CLASS TRUSS//////////////////////////

                                                                    40
dbclass Truss : public Engineering_assembly
{
public:
  Truss(char *);
  Truss(Engineering_assembly *);
  Truss(Truss*);
  Truss(char*,double,double,double,GNvector,GNvector,double);
                    //CONSTRUCTOR FOR
                    //DIRECT CREATION OF AN
                    //OBJECT BY GIVING ITS                          50
                    //ATTRIBUTES
  virtual ~Truss();
  virtual void display();
  virtual void set_attribute_values();

protected:
  virtual void propagate_attribute_values();
```

```
  virtual int check_attribute_values();
};
                                                                    60
//////////////////CLASS FRAME//////////////////

dbclass Frame : public Engineering_assembly
{
public:
  Frame(char *);
  Frame(Engineering_assembly *);
  Frame(Frame*);
  Frame(char*,double,double,double,GNvector,GNvector,double);
                    //CONSTRUCTOR FOR                               70
                    //DIRECT CREATION OF AN
                    //OBJECT BY GIVING ITS
                    //ATTRIBUTES
  virtual ~Frame();
  virtual void display();
  virtual void set_attribute_values();

protected:
  virtual void propagate_attribute_values();
  virtual int check_attribute_values();                             80
};

////////////////////////////////////////////////

#endif ENGINEERING_ASSEMBLY
```

```
#ifndef ENGINEERING_OBJECT
#define ENGINEERING_OBJECT
```

```
#include <stdlib.h>
#include <iostream.h>
#include "GNclass_dec.h"
#include "GNpersistent.h"
#include <E/dbStrings.h>
#include "GNvector.h"

dbclass Parametrized_object;


dbclass Engineering_object :  public Gab_object
{
public:

  //IN THE NEXT ENUMERATION THE COMPILER GETS
  //CONFUSED FOR THE CLASSES THAT THEY HAVE A
  //CONSTRUCTOR THAT TAKES NO ARGUMENTS. IF THE
  //NAME IN THE ENUMERATION IS THE SAME LIKE THE
  //CONSTRUCTOR'S NAME THEN THE COMPILER GIVES
  //AN ERROR, ALTHOUGH IN THE ENUMERATION THE
  //LETTERS ARE CAPITAL, WHILE IN THE CONSTRUCTOR
  //THE NAMES ARE SMALL. THAT IS WHY ALL THE
  //CLASSES WITH DUMMY CONSTRUCTORS HAVE
  //SLIGTHLY DIFFERENT NAMES THAN THE NAMES OF
  //THE CONSTRUCTORS.
  dbenum Object_type {ENGINEERINGOBJECT, ASSEMBLY,
        DIFFERENCE, INTERSECTION,
           CUBOID, CONE, CYLINDER,
           CIRCLE, RECTANGLE, SPHERE, CURVEDLINE,
           CURVEDSURFACE, SINGLECURVEDSURFACE,
           DOUBLECURVEDSURFACE, ENGINEERINGASSEMBLY,
           TRUSS, FRAME, STRAIGHTLINE,
           LINERECTCROSSSECTION, LINECIRCCROSSSECTION,
           LINERECTSOLID, LINERECTHOLLOW,
           LINECIRCSOLID, LINECIRCHOLLOW, TI, C, I, L,
```

```
        SURF,SURFACERECTPLATE, SURFACECIRCPLATE,
        SURFACERECTPLATESOLID, SURFACERECTPLATEHOLLOW,                    40
        SURFACECIRCPLATESOLID, SURFACECIRCPLATEHOLLOW};

Engineering_object(){}
Engineering_object(char *);
Engineering_object(Engineering_object*); //SSORT OF A COPY CONSTRUCTOR
Engineering_object(char*,double,double,double,GNvector,GNvector,double);
                    //CONSTRUCTOR FOR
                    //DIRECT CREATION OF AN
                    //OBJECT'S BOUNDING
                    //BOX BY GIVING ITS                                    50
                    //ATTRIBUTES
virtual ~Engineering_object();
virtual void display();
virtual void set_attribute_values();
void save();
//THE DESTRUCTOR DOES NOT DELETE THE BOUNDING BOX
//OR THE MODEL:
//SINCE WE OFTEN DELETE OBJECTS AFTER AN UPGRADE,
//AND WE DON'T LOSE EITHER THE BOUNDING BOX OR
//THE MODEL WHEN UPGRADED (WE DON'T MAKE                                   60
//COPIES OF THESE, JUST STORE THE ADDRESSES).
//HENCE THE SEPARATE EXPLICIT MECHANISM FOR
//DELETING THINGS
void delete_bounding_box();
void delete_model();
void set_bounding_box_attributes(double = 0, double = 0, double = 0);
void set_name(char *);
void set_color(char *);
void set_material(char *);
void set_specific_weight(double);                                        70
//WE DO NOT HAVE METHODS TO SET THE AREA,
//VOLUME,WEIGHT BECAUSE THEY ARE CALCULATED
//FROM THE SHAPE OF THE OBJECT AND FROM THE
//SPECIFIC WEIGHT
```

```
double calculate_area();
double calculate_volume();
double calculate_weight();

virtual void set_length(double){}
virtual void set_height(double){}
virtual void set_width(double){}
virtual void set_radius(double){}
virtual void set_number_of_divisions(int){}
virtual void set_plane(int){}
virtual void set_angle(double){}
virtual void set_chord_height(double){}
virtual void set_thickness(double){}
virtual void set_number_of_bays(int){}
virtual void set_number_of_stories(int){}
virtual void set_assembly_height(double){}
virtual void set_axis(int){}
virtual void set_foot_thickness(double){}
virtual void set_head_thickness(double){}
virtual void set_external_height(double){}
virtual void set_internal_height(double){}
virtual void set_top_width(double){}
virtual void set_bottom_width(double){}
virtual void set_x_dimension(double){}
virtual void set_y_dimension(double){}
virtual void set_z_dimension(double){}

GNmodel * get_model(){return model;}
void assign_model(GNmodel* tmp){model=tmp;}
GNcomplex* get_box(){return bounding_box;}
GNvector get_translation(){return translation;}
GNvector get_rotation(){return rotation;}
```

```
int get_object_type(){return object_type;}

void add_to_translation_vector(GNvector&);
void add_to_rotation_vector(GNvector&);
void translate(GNvector&);
void rotate(GNvector&);

void display_message(char *);
char* read_value();
                                                                    120
//OVERLOADED OPERATORS:
//UNION:
Engineering_object& operator+(Engineering_object&);
//DIFFERENCE:
Engineering_object& operator-(Engineering_object&);
//INTERSECTION:
Engineering_object& operator*(Engineering_object&);



GNset<Engineering_object*>& get_contained_objects(){               130
  return *((GNset<Engineering_object*>*)&contained_objects);
}

private:
//THE FOLLOWING GNSETS ARE FOR STORING THE
//SIMPLE OBJECTS THAT ASSEMBLIES, DIFFERENCES
//AND INTERSECTIONS COME FROM.
//THEY ARE USED MAINLY IN
//THE PARAMETRIZED OBJECT CLASS AND FOR
//KEEPING TRACK OF THE HISTORY OF EACH OBJECT.                      140
//THE CONTAINED OBJECTS CAN BE ALSO ASSEMBLIES

//THE GNSET HAS BEEN MADE PRIVATE BECAUSE IT
//IS NOT INHERITED TO THE CHILDREN CLASSES.
//IT IS USED ONLY BY THIS CLASS FOR STORING
//ASSEMBLIES.
```

```
  GNset<Engineering_object*> contained_objects;


                                                                    150
protected:
  dbchar color[50]; //HARDCODED DIMENSIONS TO KEEP
                    //IT SIMPLE
  dbchar material[50];
  dbchar name[50];
  dbdouble area;
  dbdouble volume;
  dbdouble weight;
  dbdouble specific_weight;
  GNmodel *model;                                                   160
  //THE FOLLOWING TWO GNVECTORS DEFINE THE POSITION
  //OF THE OBJECT. THEY STORE THE TOTAL TRANSLATION
  //AND ROTATION OF THE OBJECT FROM THE BEGINNING WHEN
  //IT WAS CREATED AT THE BEGINNING OF THE AXES

  GNvector translation;
  GNvector rotation;

  dbint object_type;
  dbchar wire_solid[70];                                            170
  GNcomplex *bounding_box;
  dbdouble bounding_box_x_dim;
  dbdouble bounding_box_y_dim;
  dbdouble bounding_box_z_dim;
  dbdouble flag_model_created;
  //THE FOLLOWING METHODS ARE PROTECTED BECAUSE THEY
  //ARE USED ONLY BY THE PROGRAM AND THEY CANNOT BE
  //USED BY THE USER
  virtual void propagate_attribute_values();
  virtual int check_attribute_values();                             180
  void set_wire_solid(char *);
};
```

```
#endif ENGINEERING_OBJECT
```

190

```
#ifndef GAB_OBJECT
#define GAB_OBJECT

#include <stdlib.h>
#include <iostream.h>
#include "GNclass_dec.h"
#include "GNpersistent.h"
#include <E/dbStrings.h>
#include "GNvector.h"
```

10

```
dbclass Engineering_object;

dbclass Gab_object
{

public:
  Gab_object();
  ~Gab_object();

  Engineering_object *object;
  dbdouble length;//LENGTH OF THE BOUNDING BOX
                  //OF THE OBJECT
  dbdouble width; //WIDTH OF THE BOUNDING BOX
                  //OF THE OBJECT
  dbdouble height;//HEIGHT OF THE BOUNDING BOX
```

20

```
        //OF THE OBJECT
GNvector rotation;
GNvector translation;


                                                                    30
dbenum Object_type {ENGINEERINGOBJECT, ASSEMBLY,
      DIFFERENCE, INTERSECTION,
         CUBOID, CONE, CYLINDER,
         CIRCLE, RECTANGLE, SPHERE, CURVEDLINE,
         CURVEDSURFACE, SINGLECURVEDSURFACE,
         DOUBLECURVEDSURFACE, ENGINEERINGASSEMBLY,
         TRUSS, FRAME, STRAIGHTLINE,
         LINERECTCROSSSECTION, LINECIRCCROSSSECTION,
         LINERECTSOLID, LINERECTHOLLOW,
         LINECIRCSOLID, LINECIRCHOLLOW, TI, C, I, L,                40
         SURF,SURFACERECTPLATE, SURFACECIRCPLATE,
         SURFACERECTPLATESOLID, SURFACERECTPLATEHOLLOW,
         SURFACECIRCPLATESOLID, SURFACECIRCPLATEHOLLOW};


Object_type object_type;



void set_length(double);                                            50
void set_width(double);
void set_height(double);
void set_rotation(GNvector);
void set_translation(GNvector);
void set_object_type(Object_type);

double get_length();
double get_width();
double get_height();
GNvector get_rotation();                                            60
GNvector get_translation();
```

```
  int get_object_type();

  void create_object(Object_type,char*);
  //THE ABOVE METHOD SHOULD ALSO TAKE CARE TO
  //COPY THE ATTRIBUTES OF THE 'GAB_OBJECT' CLASS
  //TO THE NEW OBJECT THAT IS BEING CREATED.


  void upgrade(Object_type);                                          70
  //THIS METHOD UPGRADES THE OBJECT TO THE NEXT LEVEL
  //WITHOUT DISPLAYING IT. IT JUST CHANGES ITS TYPE.
  //THE Object_type IS THE NEW UPGRADED TYPE


  //IT DISPLAYS THE OBJECT
  void display();

};
                                                                      80
#endif GAB_OBJECT
```

```
#include <E/collection.h>
#include "Engineering_object.h"
#include "Parametrized_object.h"
persistent extern collection<Engineering_object> objectC;
persistent extern collection<Parametrized_object> objectP;
```

```
#ifndef PARAMETRIZED_OBJECT
#define PARAMETRIZED_OBJECT

#include <stdlib.h>
#include <iostream.h>
#include "GNclass_dec.h"
#include "GNpersistent.h"
#include <E/dbStrings.h>
```

```
#include "GNvector.h"

dbclass Engineering_object;


dbclass Parametrized_object{

  friend dbclass Engineering_object;

public:
  //WE USE THIS CONSTRUCTOR WHEN WE WANT
  //TO PARAMETRIZE AN OBJECT. THE FIRST
  //ARGUMENT IS THE ADDRESS OF THE OBJECT THAT
  //WE WILL PARAMETRIZE AND THE SECOND ARGUMENT
  //IS THE NAME THAT THE PARAMETRIZED OBJECT
  //WILL TAKE.
  Parametrized_object(Engineering_object *,char *);

  ~Parametrized_object(){}

  void set_all_parameters();
  void set_one_parameter(int,int,int,double);

  //THIS METHOD TAKES AS ARGUMENT THE NAME OF
  //A PARAMETRIZED OBJECT AND IT FINDS ITS ADDRESS.
  //THEN THIS ADDRESS IS PASSED TO THE METHOD
  //ccreate_obj_from_param WHICH ASKS THE USER THE
  //DIMENSIONS AND THEN CREATES AN ENGINEERING
  //OBJECT FROM THE PARAMETRIZED ONE.
  void find_parametrized_from_name(char *);

  //THIS METHOD CREATES AN ENGINEERING OBJECT FROM A
  //PARAMETRIZED OBJECT. IT TAKES AS ARGUMENT
  //THE ADDRESS OF THE PARAMETRIZED OBJECT
  void create_obj_from_param(Parametrized_object*);
```

```
protected:
  void record_object(Engineering_object*);
  void record_parameters(char *,int,int);

  typedef dbstruct
   {                                                          50
     dbint type;
     GNvector trans;
     GNvector rot;
     dbint parameters[8];
     dbdouble values[8];
     Engineering_object *compl;
     //THE ABOVE OBJECT IS FOR
     //THE CASE THAT THE ASSEMBLY THAT WILL
     //BE PARAMETRIZED CONTAINS SOME OBJECTS
     //WHICH ARE THE DIFFERENCE OR INTERSECTION                 60
     //OF OTHER OBJECTS. THESE OBJECTS ARE
     //STORED DIRECTLY AS THEY ARE
     //WITHOUT ANY CHANGE ON THEM.
   } Par_con;

  Par_con objects[100];
  dbchar parametrized_name[50];
  dbint number_of_objects;

  dbenum Object_type {ENGINEERINGOBJECT, ASSEMBLY,            70
    DIFFERENCE, INTERSECTION,
    CUBOID, CONE, CYLINDER,
    CIRCLE, RECTANGLE, SPHERE, CURVEDLINE,
    CURVEDSURFACE, SINGLECURVEDSURFACE,
    DOUBLECURVEDSURFACE, ENGINEERINGASSEMBLY,
    TRUSS, FRAME, STRAIGHTLINE,
    LINERECTCROSSSECTION, LINECIRCCROSSSECTION,
    LINERECTSOLID, LINERECTHOLLOW,
    LINECIRCSOLID, LINECIRCHOLLOW, TBEAM,
    CBEAM, IBEAM, LBEAM,                                      80
```

```
  SURF,SURFACERECTPLATE, SURFACECIRCPLATE,
  SURFACERECTPLATESOLID, SURFACERECTPLATEHOLLOW,
  SURFACECIRCPLATESOLID, SURFACECIRCPLATEHOLLOW};

dbenum attribute_type {PARAMETER,CONSTANT};
dbenum Cuboid_parameter {CUBOLENGTH,CUBOHEIGHT,
                         CUBOWIDTH};
dbenum Cone_parameter {CONERADIUS,CONEHEIGHT,
                       CONENUMBER_OF_DIVISIONS};
dbenum Cylinder_parameter {CYLIRADIUS,CYLIHEIGHT,                    90
                       CYLINUMBER_OF_DIVISIONS};
dbenum Circle_parameter {CIRCRADIUS,CIRCPLANE,
                       CIRCNUMBER_OF_DIVISIONS};
dbenum Rectangle_parameter {RECTLENGTH,RECTWIDTH,
                         RECTPLANE};
dbenum Sphere_parameter {SPHERADIUS,
                       SPHENUMBER_OF_DIVISIONS};
dbenum Curved_line_parameter{CURLRADIUS,CURLANGLE,
                       CURLNUMBER_OF_DIVISIONS};
dbenum Curved_surface_parameter {                                    100
  CURSRADIUS,CURSANGLE,CURSWIDTH,CURSNUMBER_OF_DIVISIONS};
dbenum Single_curved_surface_parameter {
  SICURADIUS,SICUANGLE,SICUWIDTH,SICUNUMBER_OF_DIVISIONS,
  SICUTHICKNESS};
dbenum Double_curved_surface_parameter {
  DOCURADIUS,DOCUANGLE,DOCUNUMBER_OF_DIVISIONS,
  DOCUTHICKNESS};
dbenum Engineering_assembly_parameter{
  ENASNUMBER_OF_BAYS,ENASNUMBER_OF_STORIES,
  ENASASSEMBLY_HEIGHT};                                              110
dbenum Truss_parameter {
  TRUSNUMBER_OF_BAYS,TRUSNUMBER_OF_STORIES,
  TRUSASSEMBLY_HEIGHT};
dbenum Frame_parameter {
  FRAMNUMBER_OF_BAYS,FRAMNUMBER_OF_STORIES,
  FRAMASSEMBLY_HEIGHT};
```

```
dbenum Engineering_object_parameter{
  ENOBBOUNDING_BOX_X_DIM,ENOBBOUNDING_BOX_Y_DIM,
  ENOBBOUNDING_BOX_Z_DIM};
dbenum Straight_line_parameter {STLILENGTH,                              120
                                STLIAXIS};
dbenum Line_rect_cross_section_parameter {
  LRCRLENGTH,LRCRAXIS,LRCRWIDTH,LRCRHEIGHT};
dbenum Line_circ_cross_section_parameter {
  LCCRLENGTH,LCCRAXIS,LCCRRADIUS,
  LCCRNUMBER_OF_DIVISIONS};
dbenum Line_rect_solid_parameter {
  LRSOLENGTH,LRSOAXIS,LRSOWIDTH,LRSOHEIGHT};
dbenum Line_rect_hollow_parameter { LRHOLENGTH,
  LRHOAXIS, LRHOWIDTH, LRHOHEIGHT,                                       130
  LRHOTHICKNESS };
dbenum Line_circ_solid_parameter {
  LCSOLENGTH,LCSOAXIS,LCSORADIUS,
  LCSONUMBER_OF_DIVISIONS};
dbenum Line_circ_hollow_parameter {
  LCHOLENGTH,LCHOAXIS,LCHORADIUS,LCHONUMBER_OF_DIVISIONS, LCHOTHICKNESS};
dbenum T_beam_parameter {
  TILENGTH,TIAXIS,TIHEIGHT,TIWIDTH,
  TIFOOT_THICKNESS,TIHEAD_THICKNESS};
dbenum C_beam_parameter {                                                140
  CLENGTH,CAXIS,CEXTERNAL_HEIGHT,CINTERNAL_HEIGHT,
  CTOP_WIDTH,CBOTTOM_WIDTH,CTHICKNESS};
dbenum I_beam_parameter {
  ILENGTH,IAXIS,IEXTERNAL_HEIGHT,IINTERNAL_HEIGHT,
  ITOP_WIDTH,IBOTTOM_WIDTH,ITHICKNESS};
dbenum L_beam_parameter {
  LLENGTH,LAXIS,LHEIGHT,LWIDTH,LTHICKNESS};
dbenum Surface_parameter {
  SURFX_DIMENSION,SURFY_DIMENSION,SURFPLANE};
dbenum Surface_rect_plate_parameter {                                    150
  SUREX_DIMENSION,SUREY_DIMENSION,SUREPLANE,
  SUREZ_DIMENSION};
```

```
 dbenum Surface_circ_plate_parameter {
  SUCIRADIUS,SUCIHEIGHT,SUCIPLANE,SUCINUMBER_OF_DIVISIONS};
 dbenum Surface_rect_plate_solid_parameter {
  SRSOX_DIMENSION,SRSOY_DIMENSION,SRSOPLANE,
  SRSOZ_DIMENSION};
 dbenum Surface_rect_plate_hollow_parameter {
  SRHOX_DIMENSION,SRHOY_DIMENSION,SRHOPLANE,
  SRHOZ_DIMENSION,SRHOTHICKNESS};                                    160
 dbenum Surface_circ_plate_hollow_parameter {
  SCHORADIUS,SCHOHEIGHT,SCHOPLANE,SCHONUMBER_OF_DIVISIONS,
  SCHOTHICKNESS};
 dbenum Surface_circ_plate_solid_parameter {
  SCSORADIUS,SCSOHEIGHT,SCSOPLANE,SCSONUMBER_OF_DIVISIONS};

};

#endif PARAMETRIZED_OBJECT
```

```
#ifndef STRAIGHT_LINE
#define STRAIGHT_LINE

#include <iostream.h>
#include "Engineering_object.h"

//////////////////////CLASS STRAIGHT_LINE//////////

dbclass Straight_line : public Engineering_object
{                                                                    10
public:
 Straight_line(){}
 Straight_line(char *);
 Straight_line(Engineering_object *); //UPGRADE CONSTRUCTOR
 Straight_line(Straight_line*); //COPY CONSTRUCTOR
 Straight_line(char*,double,int,GNvector,GNvector,double);
                    //CONSTRUCTOR FOR
```

```
                    //DIRECT CREATION OF AN
                    //OBJECT BY GIVING ITS
                    //ATTRIBUTES                                    20
  virtual ~Straight_line();
  virtual void display();
  virtual void set_attribute_values();
  virtual void set_length(double);
  virtual void set_axis(int);

protected:
  dbdouble length;
  dbint axis;
  virtual void propagate_attribute_values();                        30
  virtual int check_attribute_values();
};

//////////CLASS LINE_RECT_CROSS_SECTION/////////////

dbclass Line_rect_cross_section : public Straight_line
{
public:
  Line_rect_cross_section(){}
  Line_rect_cross_section(char *);                                  40
  Line_rect_cross_section(Straight_line *);
  Line_rect_cross_section(Line_rect_cross_section *);
  Line_rect_cross_section(char*,double,int,double,double,GNvector,GNvector,double);
                    //CONSTRUCTOR FOR
                    //DIRECT CREATION OF AN
                    //OBJECT BY GIVING ITS
                    //ATTRIBUTES
  virtual ~Line_rect_cross_section();
  virtual void display();
  virtual void set_attribute_values();                              50
  virtual void set_width(double);
  virtual void set_height(double);
```

```
protected:
  dbdouble width;
  dbdouble height;
  virtual void propagate_attribute_values();
  virtual int check_attribute_values();
};

                                                                          60
//////////////CLASS LINE_CIRC_CROSS_SECTION//////////

dbclass Line_circ_cross_section : public Straight_line
{
public:
  Line_circ_cross_section(){}
  Line_circ_cross_section(char *);
  Line_circ_cross_section(Line_circ_cross_section *);
  Line_circ_cross_section(Straight_line *);
  Line_circ_cross_section(char*,double,int,double,int,GNvector,GNvector,double);   70
                    //CONSTRUCTOR FOR
                    //DIRECT CREATION OF AN
                    //OBJECT BY GIVING ITS
                    //ATTRIBUTES
  virtual ~Line_circ_cross_section();
  virtual void display();
  virtual void set_attribute_values();
  virtual void set_radius(double);
  virtual void set_number_of_divisions(int);

                                                                          80
protected:
  dbdouble radius;
  dbint number_of_divisions;
  virtual void propagate_attribute_values();
  virtual int check_attribute_values();
};

//////////////CLASS LINE_RECT_SOLID/////////////////
```

```
dbclass Line_rect_solid : public Line_rect_cross_section                    90
{
public:
  Line_rect_solid(char *);
  Line_rect_solid(Line_rect_cross_section *);
  Line_rect_solid(Line_rect_solid*);
  Line_rect_solid(char*,double,int,double,double,GNvector,GNvector,double);
                    //CONSTRUCTOR FOR
                    //DIRECT CREATION OF AN
                    //OBJECT BY GIVING ITS
                    //ATTRIBUTES                                            100
  virtual ~Line_rect_solid();
  virtual void display();
  virtual void set_attribute_values();

protected:
  virtual void propagate_attribute_values();
  virtual int check_attribute_values();
};

//////////////CLASS LINE_RECT_HOLLOW/////////////////                        110

dbclass Line_rect_hollow : public Line_rect_cross_section
{
public:
  Line_rect_hollow(char *);
  Line_rect_hollow(Line_rect_cross_section *);
  Line_rect_hollow(Line_rect_hollow*);
  Line_rect_hollow(char*,double,int,double,double,double,GNvector,GNvector,double);
                    //CONSTRUCTOR FOR
                    //DIRECT CREATION OF AN                                 120
                    //OBJECT BY GIVING ITS
                    //ATTRIBUTES
  virtual ~Line_rect_hollow();
  virtual void display();
  virtual void set_attribute_values();
```

```
  virtual void set_thickness(double);

protected:
  dbdouble thickness;
  virtual void propagate_attribute_values();                                    130
  virtual int check_attribute_values();
};

////////////////CLASS LINE_CIRC_SOLID/////////////

dbclass Line_circ_solid : public Line_circ_cross_section
{
public:
  Line_circ_solid(char *);
  Line_circ_solid(Line_circ_cross_section *);                                   140
  Line_circ_solid(Line_circ_solid *);
  Line_circ_solid(char*,double,int,double,int,GNvector,GNvector,double);
                    //CONSTRUCTOR FOR
                    //DIRECT CREATION OF AN
                    //OBJECT BY GIVING ITS
                    //ATTRIBUTES
  virtual ~Line_circ_solid();
  virtual void display();
  virtual void set_attribute_values();
                                                                                150
protected:
  virtual void propagate_attribute_values();
  virtual int check_attribute_values();
};

//////////////////CLASS LINE_CIRC_HOLLOW/////////
dbclass Line_circ_hollow : public Line_circ_cross_section
{
public:
  Line_circ_hollow(char *);                                                     160
  Line_circ_hollow(Line_circ_cross_section *);
```

```
  Line_circ_hollow(Line_circ_hollow *);
  Line_circ_hollow(char*,double,int,double,int,double,GNvector,GNvector,double);
                    //CONSTRUCTOR FOR
                    //DIRECT CREATION OF AN
                    //OBJECT BY GIVING ITS
                    //ATTRIBUTES
  virtual ~Line_circ_hollow();
  virtual void display();
  virtual void set_attribute_values();                                    170
  virtual void set_thickness(double);

protected:
  dbdouble thickness;
  virtual void propagate_attribute_values();
  virtual int check_attribute_values();
};

//////////////CLASS T_beam/////////////////////////////////
                                                                          180
dbclass T_beam : public Straight_line
{
public:
  T_beam(char *);
  T_beam(Straight_line *);
  T_beam(T_beam*); //  copy constructor
  T_beam(char*,double,int,double,double,double,double,GNvector,GNvector,double);
                    //CONSTRUCTOR FOR
                    //DIRECT CREATION OF AN
                    //OBJECT BY GIVING ITS                              190
                    //ATTRIBUTES
  virtual ~T_beam();
  virtual void display();
  virtual void set_attribute_values();
  virtual void set_height(double);
  virtual void set_width(double);
  virtual void set_foot_thickness(double);
```

```
  virtual void set_head_thickness(double);

protected:                                                          200
  dbdouble height;
  dbdouble width;
  dbdouble foot_thickness;
  dbdouble head_thickness;
  virtual void propagate_attribute_values();
  virtual int check_attribute_values();
};

//////////////////////CLASS C_beam/////////////////////
                                                                    210
dbclass C_beam : public Straight_line
{
public:
  C_beam(char *);
  C_beam(Straight_line *);
  C_beam(C_beam*) ; //  copy constructor
  C_beam(char*,double,int,double,double,double,double,double,GNvector,GNvector,double);
                    //CONSTRUCTOR FOR
                    //DIRECT CREATION OF AN
                    //OBJECT BY GIVING ITS                          220
                    //ATTRIBUTES
  virtual ~C_beam();
  virtual void display();
  virtual void set_attribute_values();
  virtual void set_external_height(double);
  virtual void set_internal_height(double);
  virtual void set_top_width(double);
  virtual void set_bottom_width(double);
  virtual void set_thickness(double);
                                                                    230
protected:
  dbdouble external_height;
  dbdouble internal_height;
```

```
  dbdouble top_width;
  dbdouble bottom_width;
  dbdouble thickness;
  virtual void propagate_attribute_values();
  virtual int check_attribute_values();
};
                                                                    240
//////////////////CLASS I_beam//////////////////////////
dbclass I_beam : public Straight_line
{
public:
  I_beam(char *);
  I_beam(Straight_line *);
  I_beam(I_beam*);
  I_beam(char*,double,int,double,double,double,double,double,GNvector,GNvector,double);
                    //CONSTRUCTOR FOR
                    //DIRECT CREATION OF AN                         250
                    //OBJECT BY GIVING ITS
                    //ATTRIBUTES
  virtual ~I_beam();
  virtual void display();
  virtual void set_attribute_values();
  virtual void set_external_height(double);
  virtual void set_internal_height(double);
  virtual void set_top_width(double);
  virtual void set_bottom_width(double);
  virtual void set_thickness(double);                               260

protected:
  dbdouble external_height;
  dbdouble internal_height;
  dbdouble top_width;
  dbdouble bottom_width;
  dbdouble thickness;
  virtual void propagate_attribute_values();
  virtual int check_attribute_values();
```

```
};                                                                          270

//////////////////////CLASS L_beam/////////////////////

dbclass L_beam : public Straight_line
{
public:
  L_beam(char *);
  L_beam(Straight_line *);
  L_beam(L_beam*);
  L_beam(char*,double,int,double,double,double,GNvector,GNvector,double);   280
                    //CONSTRUCTOR FOR
                    //DIRECT CREATION OF AN
                    //OBJECT BY GIVING ITS
                    //ATTRIBUTES
  virtual ~L_beam();
  virtual void display();
  virtual void set_attribute_values();
  virtual void set_height(double);
  virtual void set_width(double);
  virtual void set_thickness(double);                                       290

protected:
  dbdouble height;
  dbdouble width;
  dbdouble thickness;
  virtual void propagate_attribute_values();
  virtual int check_attribute_values();
};

#endif STRAIGHT_LINE                                                        300
```

```
#ifndef SURFACE
#define SURFACE

#include <iostream.h>
#include "Engineering_object.h"

//////////CLASS SURFACE//////////////////////

dbclass Surface : public Engineering_object
{                                                                    10
public:
  Surface(){}
  Surface(char *);
  Surface(Engineering_object *); //UPGRADE CONSTRUCTOR
  Surface(Surface*); //COPY CONSTRUCTOR
  Surface(char*,double,double,int,GNvector,GNvector,double);
                    //CONSTRUCTOR FOR
                    //DIRECT CREATION OF AN
                    //OBJECT BY GIVING ITS
                    //ATTRIBUTES                                     20
  virtual ~Surface();
  virtual void display();
  virtual void set_attribute_values();
  virtual void set_x_dimension(double);
  virtual void set_y_dimension(double);
  virtual void set_plane(int);
```

```
protected:
  dbdouble x_dimension;
  dbdouble y_dimension;                                                    30
  dbint plane;
  virtual void propagate_attribute_values();
  virtual int check_attribute_values();
};

//////////////CLASS SURFACE_RECT_PLATE/////////////

dbclass Surface_rect_plate : public Surface
{
public:                                                                    40
  Surface_rect_plate(){}
  Surface_rect_plate(char *);
  Surface_rect_plate(Surface *);
  Surface_rect_plate(Surface_rect_plate *);
  Surface_rect_plate(char*,double,double,int,double,GNvector,GNvector,double);
                    //CONSTRUCTOR FOR
                    //DIRECT CREATION OF AN
                    //OBJECT BY GIVING ITS
                    //ATTRIBUTES
  virtual ~Surface_rect_plate();                                           50
  virtual void display();
  virtual void set_attribute_values();
  virtual void set_z_dimension(double);

protected:
  dbdouble z_dimension;
  virtual void propagate_attribute_values();
  virtual int check_attribute_values();
};
                                                                           60
//////////////CLASS SURFACE_CIRC_PLATE/////////////

dbclass Surface_circ_plate : public Surface
```

```
{
public:
  Surface_circ_plate(){}
  Surface_circ_plate(char *);
  Surface_circ_plate(Surface *);
  Surface_circ_plate(Surface_circ_plate*);
  Surface_circ_plate(char*,double,double,int,int,GNvector,GNvector,double);
                    //CONSTRUCTOR FOR
                    //DIRECT CREATION OF AN
                    //OBJECT BY GIVING ITS
                    //ATTRIBUTES
  virtual ~Surface_circ_plate();
  virtual void display();
  virtual void set_attribute_values();
  virtual void set_radius(double);
  virtual void set_height(double);
  virtual void set_number_of_divisions(int);

protected:
  dbdouble radius;
  dbdouble height;
  //  THE VARIABLE height IS THE THICKNESS OF
  //THE PLATE. WE GIVE THE NAME HEIGHT
  //BECAUSE WE USE THICKNESS FOR THE HOLLOW
  //CIRCULAR PLATE
  dbint number_of_divisions;
  virtual void propagate_attribute_values();
  virtual int check_attribute_values();
};

//////////CLASS SURFACE_RECT_PLATE_SOLID//////////

dbclass Surface_rect_plate_solid : public Surface_rect_plate
{
public:
  Surface_rect_plate_solid(char *);
```

```
  Surface_rect_plate_solid(Surface_rect_plate *);                                    100
  Surface_rect_plate_solid(Surface_rect_plate_solid *);
  Surface_rect_plate_solid(char*,double,double,int,double,GNvector,GNvector,double);
                        //CONSTRUCTOR FOR
                        //DIRECT CREATION OF AN
                        //OBJECT BY GIVING ITS
                        //ATTRIBUTES
  virtual ~Surface_rect_plate_solid();
  virtual void display();
  virtual void set_attribute_values();
                                                                                     110
protected:
  virtual void propagate_attribute_values();
  virtual int check_attribute_values();
};

//////////////CLASS SURFACE_RECT_PLATE_HOLLOW//////
dbclass Surface_rect_plate_hollow : public Surface_rect_plate
{
public:
  Surface_rect_plate_hollow(char *);                                                 120
  Surface_rect_plate_hollow(Surface_rect_plate *);
  Surface_rect_plate_hollow(Surface_rect_plate_hollow *);
  Surface_rect_plate_hollow(char*,double,double,int,double,double,GNvector,GNvector,double);
                        //CONSTRUCTOR FOR
                        //DIRECT CREATION OF AN
                        //OBJECT BY GIVING ITS
                        //ATTRIBUTES
  virtual ~Surface_rect_plate_hollow();
  virtual void display();
  virtual void set_attribute_values();                                               130
  virtual void set_thickness(double);

protected:
  dbdouble thickness;
  virtual void propagate_attribute_values();
```

```
  virtual int check_attribute_values();
};
////////////////////////////////////////////////
                                                                    140

//////////CLASS SURFACE_CIRC_PLATE_HOLLOW/////////////

dbclass Surface_circ_plate_hollow : public Surface_circ_plate
{
public:
  Surface_circ_plate_hollow(char *);
  Surface_circ_plate_hollow(Surface_circ_plate *);
  Surface_circ_plate_hollow(Surface_circ_plate_hollow *);
  Surface_circ_plate_hollow(char*,double,double,int,int,double,GNvector,GNvector,double); 150
                    //CONSTRUCTOR FOR
                    //DIRECT CREATION OF AN
                    //OBJECT BY GIVING ITS
                    //ATTRIBUTES
  virtual ~Surface_circ_plate_hollow();
  virtual void display();
  virtual void set_attribute_values();
  virtual void set_thickness(double);

protected:                                                          160
  dbdouble thickness;
  virtual void propagate_attribute_values();
  virtual int check_attribute_values();
};


/////////CLASS SURFACE_CIRC_PLATE_SOLID//////////////
dbclass Surface_circ_plate_solid : public Surface_circ_plate
{
public:                                                             170
  Surface_circ_plate_solid(char *);
```

```
  Surface_circ_plate_solid(Surface_circ_plate *);
  Surface_circ_plate_solid(Surface_circ_plate_solid *);
  Surface_circ_plate_solid(char*,double,double,int,int,GNvector,GNvector,double);
                    //CONSTRUCTOR FOR
                    //DIRECT CREATION OF AN
                    //OBJECT BY GIVING ITS
                    //ATTRIBUTES
  virtual ~Surface_circ_plate_solid();
  virtual void display();                                                    180
  virtual void set_attribute_values();

protected:
  virtual void propagate_attribute_values();
  virtual int check_attribute_values();
};

#endif SURFACE

                                                                             190

                                                                             200
```

Appendix C

APPENDIX C: SHAPE DIMENSIONS

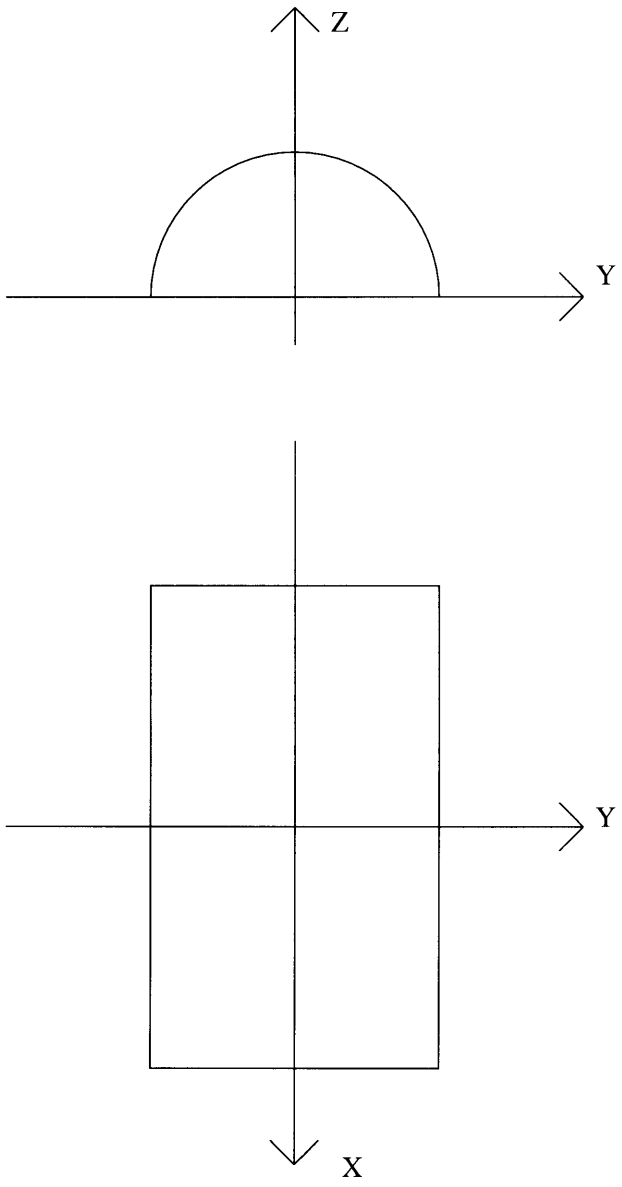

Figure C-1: Curved Surface

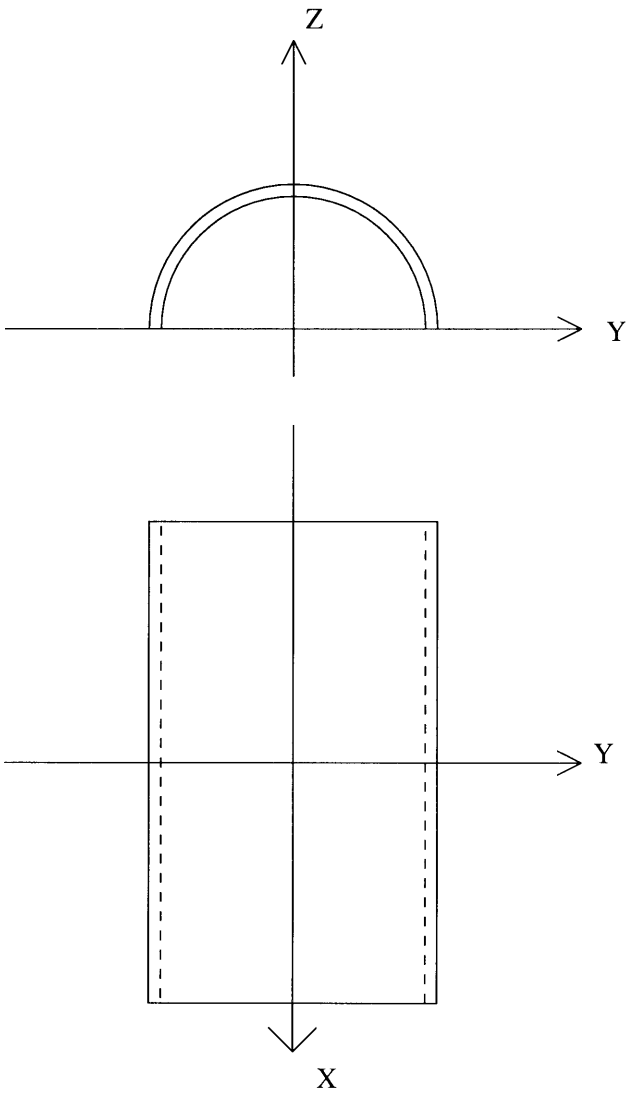

Figure C-2: Single Curved Surface

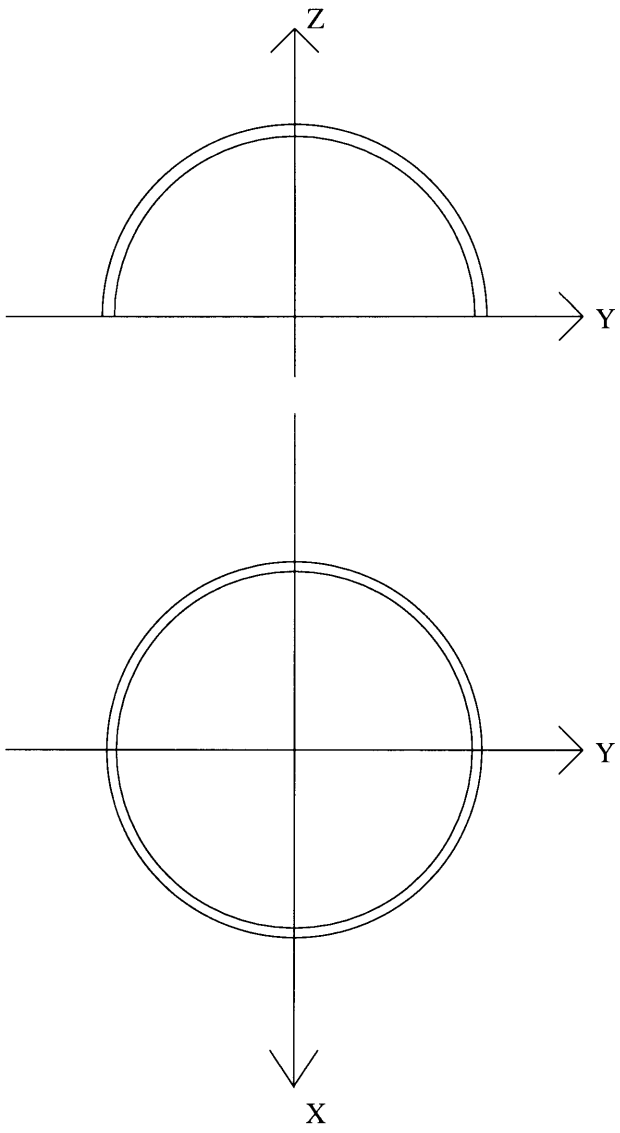

Figure C-3: Double Curved Surface

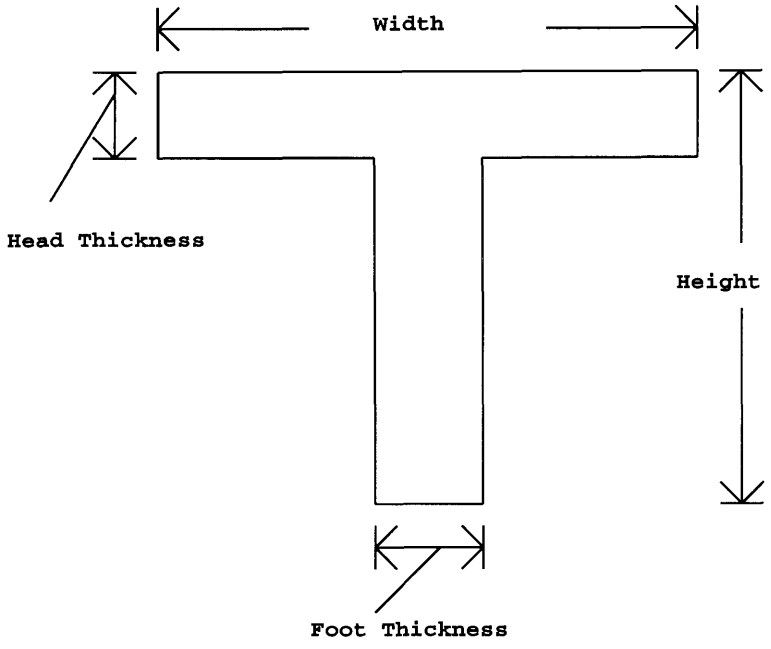

Figure C-4: T Beam Dimensions

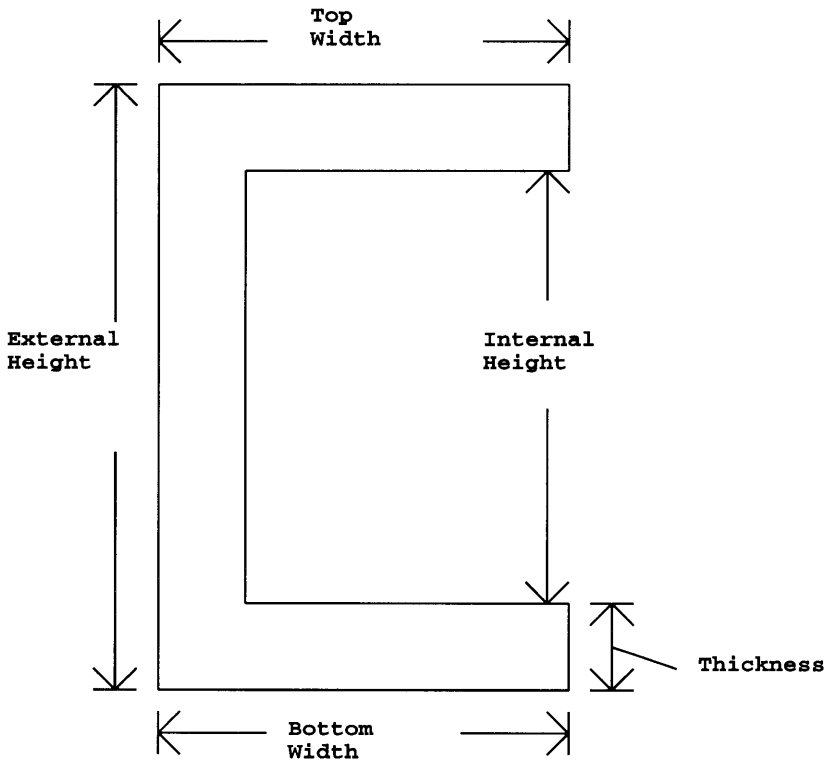

Figure C-5: C Beam Dimensions

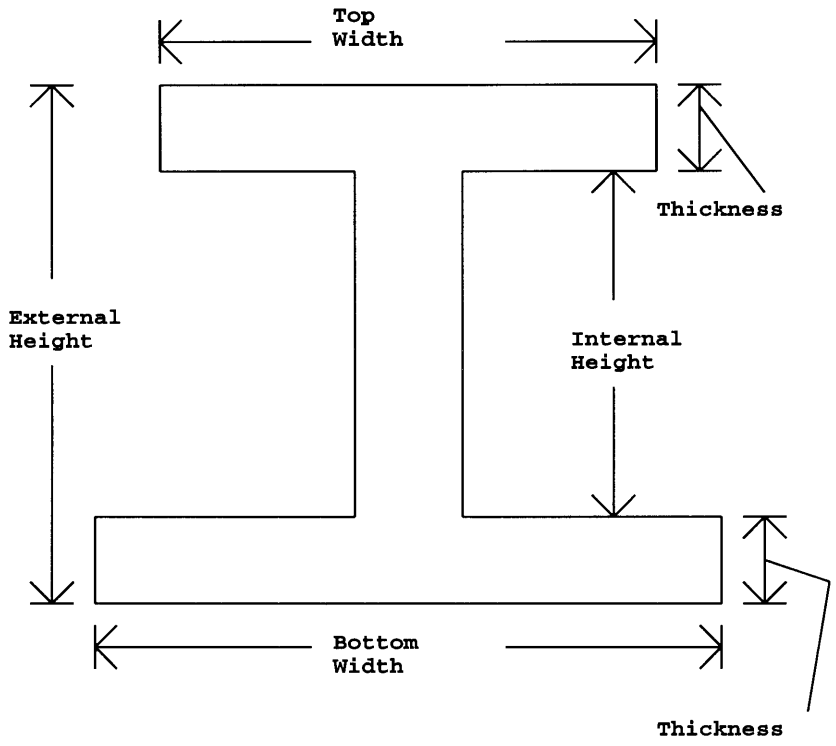

Figure C-6: I Beam Dimensions

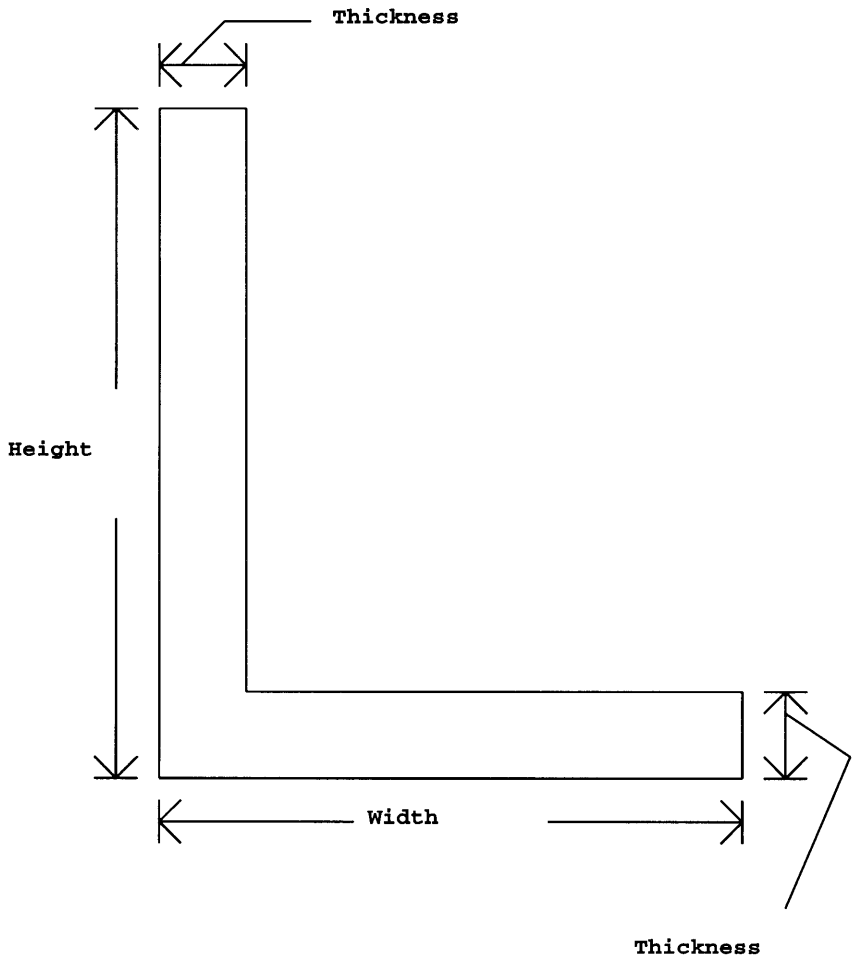

Figure C-7: L Beam Dimensions

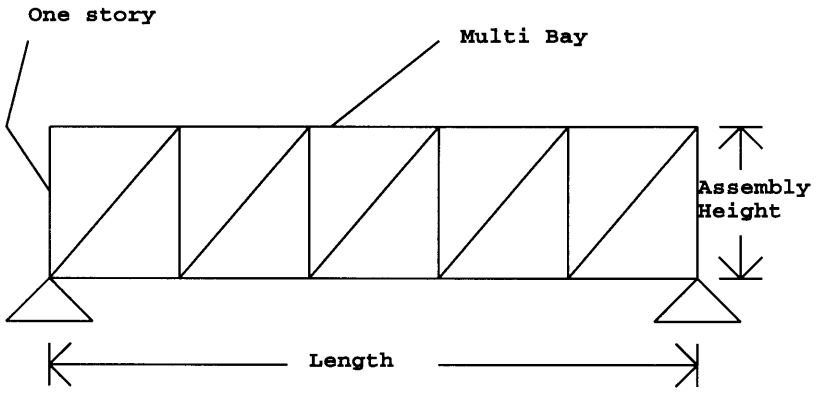

Figure C-8: Truss Dimensions

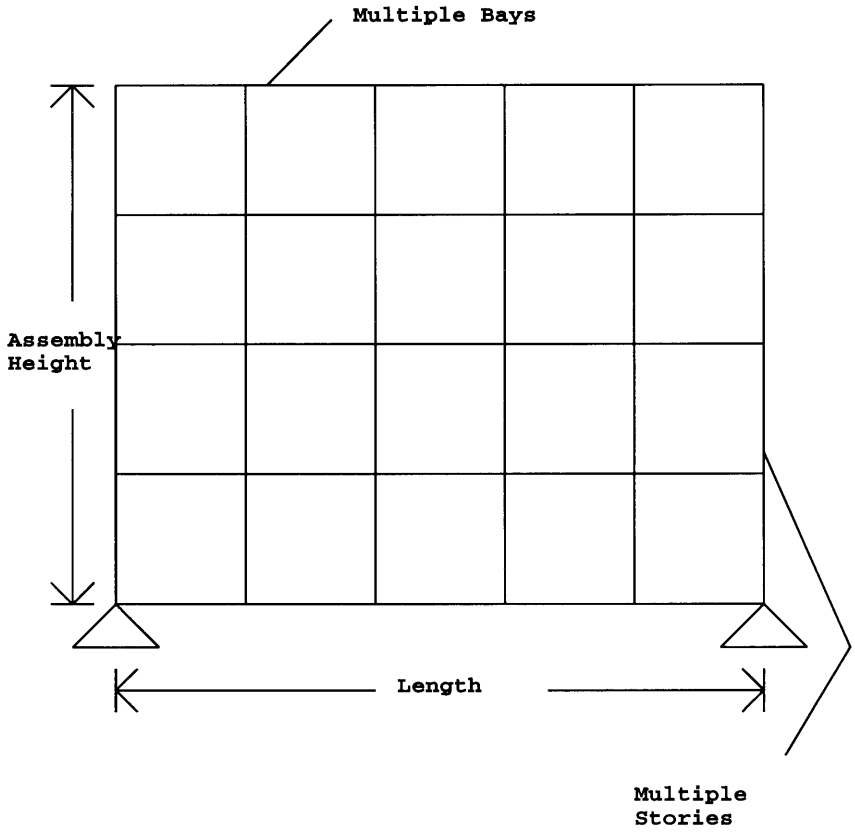

Figure C-9: Frame Dimensions

Appendix D

APPENDIX D: EXAMPLES

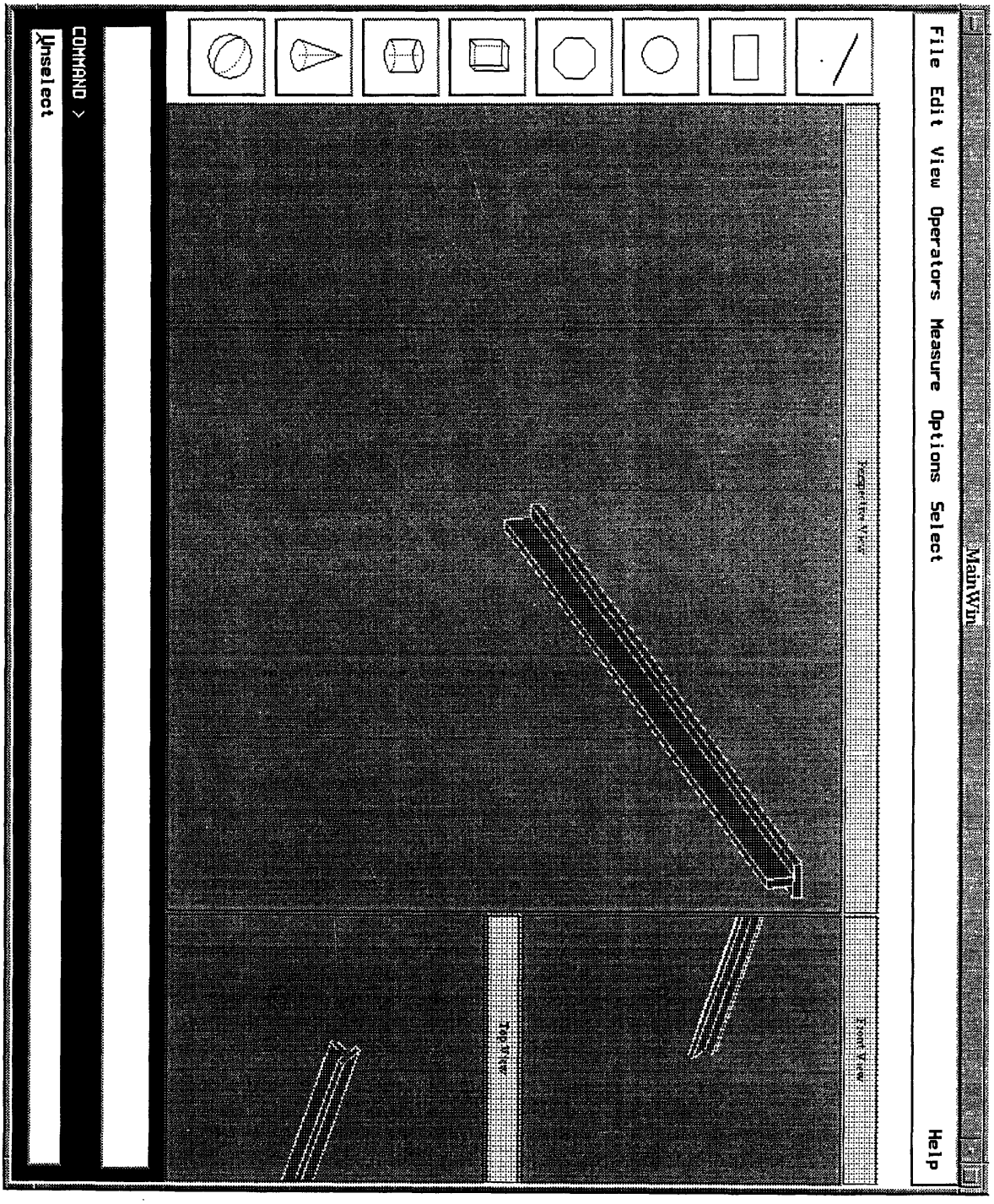

Figure D-1: T Beam Example

Figure D-2: C Beam Example

Figure D-3: I Beam Example

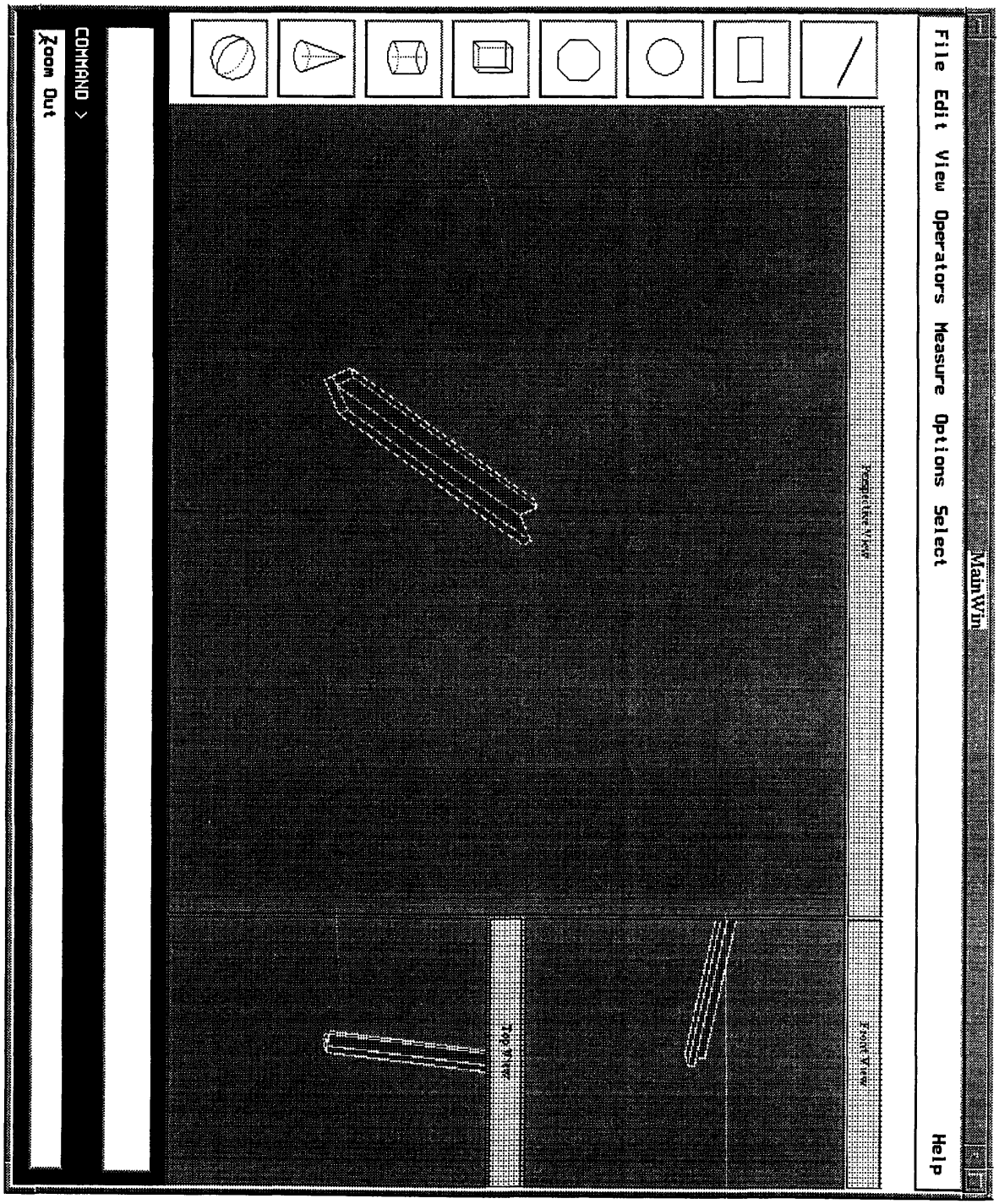

Figure D-4: L Beam Example

Bibliography

[1] M. Carey, D. Dewitt, and E. Shekita. *Storage Management for Objects in EXODUS, Object-Oriented Concepts, Databases, and Applications.* W. Kim and F. Lochovsky, eds., Addison-Wesley, 1989.

[2] J. J. Cunningham and J. R. Dixon. Designing with features: the origin. In *Design Theory 88, Proceedings of the NSF Grantees Workshop in Engineering Design, RPI*, Troy, New York, June 1988. Springer Verlag.

[3] T. Gaskins. *PHIGS Programming Manual.* O'Reilly & Associates, Inc, 1992.

[4] A. Goldman, K. Hussein, G. Margelis, J. Sugiono, and Y. C. Huang. GRAPHITI Detailed Design. Project for Course: *Computer Aided Engineering II: Software Engineering for CAE Systems*, 1.552 of M.I.T., May 1994.

[5] A. Goldman, K. Hussein, G. Margelis, J. Sugiono, and Y. C. Huang. GRAPHITI Functional Specifications. Project for Course *Computer Aided Engineering II: Software Engineering for CAE Systems*, 1.552 of M.I.T., April 1994.

[6] S. R. Gorti. From Symbol to Form: A Framework for Design Evolution. Doctoral Research Proposal, Intelligent Engineering Systems Laboratory, Department of Civil and Environmental Engineering, M.I.T., May 1993. The document can be obtained by: *ftp iesl.mit.edu, login name = anonymous*, under the pathname: */pub/dice/congen/GONGEN.ps.*

[7] S. R. Gorti and D. Sriram. CONGEN: An Integrated Approach to Conceptual Design. *International Journal of CAD/CAM and Computer Graphics*, 8(2):135–150, 1993.

[8] Li-Xing He. A Non-Manifold Geometry Modeler: An Object Oriented Approach. Master's thesis, M.I.T., 1993.

[9] C. M. Hoffmann. *Geometric and Solid Modeling: An Introduction.* Morgan Kaufmann Publishers, Inc, 1989.

[10] T. L. J. Howard. Evaluating PHIGS for CAD and general graphics applications. *Computer Aided Design*, 23(4):244–251, 1991.

[11] Ithaca Software, Alameda CA. *Hoops Graphics System, Reference Manual, Version 2.2*, 1990.

[12] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice Hall, 1988.

[13] T. Laako and M. Mantyla. A feature definition language for bridging solid and features. In J. Rossignac, J. Turner, and G. Allen, editors, *2nd ACM Solid Modeling Conference*, pages 333–342. ACM Press, May 1993.

[14] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The Objectstore Database System. In *Comm. ACM*, volume 34(10), Oct 1991.

[15] P. J. Lucas. *The C++ Programmer's Handbook.* Prentice Hall, 1992.

[16] M. Mantyla. *An Introduction to Solid Modeling.* Computer Science Press, 1988.

[17] M. Mantyla. A modeling system for top-down design of assembled products. *IBM Journal of Research and Development*, 34(5):636–659, 1990.

[18] M. E. Mortenson. *Geometric Modeling.* John Wiley & Sons, New York, 1985.

[19] O'Reilly & Associates, Inc. *PHIGS Reference Manual*, 1992.

[20] J. E. Richardson. *E: A Persistent Systems Implementation Language.* PhD thesis, University of Wisconsin, Madison, August 1989.

[21] J. Rossignac and M. O'Connor. Selective Geometric Complexes: A Dimension-Independent Model for Representing Point Sets with Internal Structures and Incomplete Boundaries. M. Wozny, J. U. Turner, and K. Preiss, editors, North-Holland, 1990.

[22] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[23] D. L. Schodek. *Structures.* Englewood Cliffs, N.J.: Prentice-Hall, 1980.

[24] J. J. Shah. Conceptual Development of Form Features and Feature Modelers. *Research in Engineering Design*, 2:93–108, 1991.

[25] J. J. Shah and M. T. Rogers. Feature Based Modeling Shell: Design and Implementation. In *Design Theory 88, Proceedings of the NSF Grantees Workshop in Engineering Design, RPI*, Troy, New York, June 1988. Springer Verlag.

[26] T. Smithers. AI-based design versus geometry-based design or Why design cannot be supported by geometry alone. *Computer Aided Design*, 21(3):141–150, 1989.

[27] M. Stefic and D.G. Bobrow. Object-Oriented Programming: Themes and Variation. *AI Magazine*, 6(4):40–62, 1986.

[28] R. S. Wiener and L. J. Pinson. *The C++ Workbook.* Addison-Wesley Publishing Company, 1990.

[29] A. Wong and D. Sriram. Design Document for the GNOMES Geometric Modeler. IESL Technical Report, Intelligent Engineering Systems Laboratory, M.I.T., October 1991. The document can be obtained by: *ftp iesl.mit.edu, login name = anonymous*, under the pathname: */pub/dice/gnomes/GNOMES_design_doc.ps.*

[30] A. Wong and D. Sriram. Geometric Modeling for Cooperative Product Development. Order No. IESL 92-04, Research Report 92-28, October 1992. The report can be obtained by: *ftp iesl.mit.edu, login name = anonymous*, under the pathname: */pub/dice/gnomes/GNOMES_paper.ps.*

[31] R. F. Woodbury and I. J. Oppenheim. An Approach to Geometric Reasoning. In *Intelligent CAD, I, Proceedings of the IFIP TC 5/WG 5.2 Workshop on Intelligent CAD, North-Holland*, 1987.

[32] D. A. Young. *Object-Oriented Programming with C++ and OSF/Motif.* Prentice Hall, 1992.