Development of an Object-Oriented Graphical User Interface for an Aircraft Engine Cycle Analysis Program

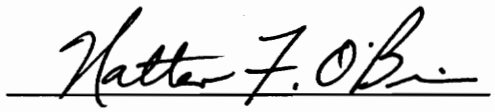
by

Andreas Steude

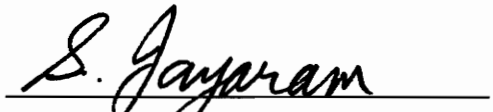
Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Mechanical Engineering

APPROVED:



Dr. Walter F. O'Brien, Chairman



Dr. S. Jayaram

Dr. A. Myklebust

July 19, 1993
Blacksburg, Virginia

Abstract

Since the 1960's an overwhelming amount of in-house and custom engineering software has been written. In the effort to reduce the cost of maintaining existing codes and producing new applications, the recent introduction of the object-oriented design approach has proved successful. At the same time graphical user interfaces are gaining in popularity to improve the usability and versatility of an application. This thesis investigates the application of the object-oriented approach to the design of a graphical user interface for an engineering design application. The development of an object-oriented graphical user interface for the NASA Engine Performance Program, a turbine engine design code is presented. The design of the new object-oriented graphical user interface for extensibility and re-usability is discussed. Design considerations for integration of the interface with procedural and object-oriented versions of the conceptual aircraft design program, ACSYNT, are explained. An existing PHIGS-based object-oriented graphical user interfacing framework is extended and built upon to develop the class structure of the interface. The class organization is presented in commonly used notation and described in detail.

Acknowledgments

"I swear - by my life and my love for it - that I shall never live for the sake of another man, nor ask another man to live for mine."
- from the speech of John Galt, Atlas Shrugged by Ayn Rand.

It is seldom that a person can claim full credit for an achievement and I cannot make an exception in this case. I owe a debt of gratitude to a number of people who helped make this work possible through contributions of wisdom, knowledge, hard work and friendship.

I have known my current advisor, Dr Walter F. O'Brien, since my senior year as an undergraduate in the aerospace engineering curriculum. It was because of his patient confidence in my ability that I got the chance to work on several interesting projects in the Mechanical Engineering Department, completing my thesis with the current one.

Sincere thanks are also due to the other members of my graduate committee. Dr Arvid Myklebust and Dr Sankar Jayaram both provided much guidance with their wealth of knowledge in the extended field of Computer Aided Engineering. Thanks also for the insight on topics not directly related to my research.

The research was funded by NASA-Lewis, through the ACSYNT Institute, a conglomerate venture by industry, government and academia. Without the resources provided by them this work could not have been done.

My work was built on a new GUI framework written by the resident C++ god, Scott Woyak. Thanks for your patience and help when I stumbled across another bug in your code -- especially in the many cases when the bug was my own.

My most important and influential teachers are my parents. They gave me the strong foundation and constant encouragement to persevere through these past years. *Ihr seid meine besten und liebsten Vorbilder - Vielen Dank für Alles von Eurem Buben.* Thanks also to my sisters. They say you can't choose your family, but I would still pick you.

Alan - Thanks for the many discussions and arguments.

Brett and Uma - Thanks for staying so down-to-earth and keeping me in touch with reality.

John and Francisco - I couldn't have pulled the night shifts without you.

Frank, Greg, and everyone from the Turbo Lab and CAD Lab - I'd like to thank you for all the help and fun times over the course of my graduate career.

Valerie - Thanks for all the distractions: *All work and no play makes Jack a dull boy. All work and no play makes Jack a dull boy. All work and no play...*

Lastly, I would like to thank Eric V. Schrock for introducing me to the philosophy of Ayn Rand. Once I get past the inferiority complex induced by it, I hope I can live up to the principles of Objectivism.

Table of Contents

List of Illustrations

Introduction

This thesis addresses the inevitable marriage of two trends that have become evident in the computer software and engineering design community. The demand for graphical user interfaces (GUI) and the evolution of procedural programming languages and methods into object-based and object-oriented (OO) languages and methods combines to make the OO-GUI. In the present work the development of an object-oriented graphical user interface is described in its application to the aircraft engine cycle analysis program, NEPP (NASA Engine Performance Program), for inclusion with the ACSYNT aircraft conceptual design program. Both NEPP and ACSYNT are described in a later section.

In recent years the flood of computer programs has grown to unmanageable proportions. Not only are major software companies developing and releasing software at an unprecedented rate but just about every major firm has their own in-house codes. Managing the complexity of these codes has become a primary concern **[Booc91]**.

Engineering companies especially, usually have to be self-sufficient when it comes to maintaining and upgrading computer programs. This is mostly due to the degree of specialization of the codes. To a certain extent proprietary concerns play a role as well. Often the engineer takes on the task of developing, maintaining and upgrading in-house design computer programs. More and more manpower is being allocated to this task as the codes become more complex, larger and as a result less manageable. In an environment

where the order of magnitude has grown up to the million mark, it is estimated that 50,000 lines of conventional code is the most a single programmer is able to maintain **[Schi92]**.

In light of the makeup of most engineering design problems, a modular programming approach has been favored for most newer applications. Trying to model their problem in the most natural, intuitive and logical manner, many programmers have moved in the direction of object-oriented code within the limits of the procedural constraints of their programming language.

Many object-oriented or object-based languages have appeared on the market as a result of a natural evolution of programming languages **[Booc91]**. Graduating from FORTRAN, C, Pascal and LISP, engineers are now starting to take advantage of their object-oriented heirs Simula, Smalltalk, Object Pascal, C++, CLOS and Ada among others.

Most engineering design codes traditionally started out as batch programs. Facades were sometimes added in the form of line-by-line question and answer or primitive non-graphical menu driven interfaces. These user interfacing methods assume control of the application by dictating the path the user takes in executing the program.

With the growing availability of computers with graphics capabilities, a big emphasis is currently being placed on endowing existing software with graphical user interfaces. The recent push for graphical user interfaces comes with the recognition of the flexibility a well-implemented GUI lends to an application. Extensive research in human-computer interaction has resulted in a set of conventions and guidelines to be followed in implementing a GUI **[Fisc89]**. The success of applications written for environments such as Microsoft Windows, Macintosh, OS/2 or X Windows speaks for itself.

It is only to be expected that interest in the object-oriented graphical user interfaces for engineering design applications should rise simultaneously with the development of the analysis programs coded in the object-oriented paradigm. While the present work implements an OO-GUI for a traditionally coded program, NEPP, it is designed for use with object-oriented aircraft conceptual design codes such as the object-oriented ACSYNT program currently being developed at the Computer Aided Design Laboratory at the Virginia Polytechnic Institute and State University.

Objectives

The goal of this work is to design and implement a user interface for the aircraft engine cycle analysis program, NEPP, to be used with the interactive CAD aircraft conceptual design program, ACSYNT. The interface must satisfy the following objectives:

1. ***Flexibility and Ease of Use*** - By using a modern graphical user interface with a familiar (Motif-like) look and feel, the interface can be made intuitive and flexible to use.

2. ***Device Independence and Ease of Integration*** - With the use of the standard or soon-to-be standard programming languages and graphics languages, C/C++, FORTRAN77, and PHIGS, device independence and compatibility with other CAD applications can be achieved. By using the object-oriented paradigm to decompose and model the problem space in consistent classes and objects, integration with other object-oriented applications can be simplified.

3. ***Extensibility and Maintainability*** - The interface will be designed and implemented in the object-oriented paradigm. This will enhance extensibility and maintainability as explained in a later section.

Literature Survey

Object-Oriented Paradigm

Object-oriented design and object-oriented programming has become very prominent in the computer science and software engineering world in the last decade. Much experience has been gained in the field and many excellent comprehensive and specialized books and papers have been published.

Grady Booch's Object-Oriented Design With Applications **[Booc91]** has become one of the most quoted authoritative references on object-oriented design methods. It also provides samples of object-oriented design in several different programming languages. Although, as everywhere in the fast-paced world of computer science, obsolescence of the specialized material lurks just around the corner, the book provides a valuable foundation for evolving a customized software development philosophy. It also contains a thorough explanation of the notation used in this work.

Boehm provides a very detailed description of the software development process on a corporate level. The model described in "A Spiral Model of Software Development and Enhancement" is an alternative interpretation of the methods discussed by Booch. It adds

another dimension to Booch's description with the inclusion of cost and risk analysis **[Boeh86]**.

Object-orientation with respect to computer graphics has been discussed in several references. Computer Graphics Using Object-Oriented Programming, edited by Cunningham et al., is a collection of papers describing general methods for the application of object-oriented design to graphics programming **[Cunn92]**. Object-Oriented Graphics - From GKS and PHIGS to Object-Oriented Systems by Wisskirchen calls for a standardized object-oriented graphics language (or framework) and demonstrates his ideas as applied to GKS and PHIGS. He also introduces GEO++, an object-oriented graphics system **[Wiss90]**. Wampler presented his proposal for a PHIGS-based object-oriented graphics framework prototype in 1991 **[Wamp91]**. One of the most widely known and used object-oriented graphics system is HOOPS **[Murp93][HOOP92]**.

Graphical User Interfacing

A wealth of information is available on all aspects of graphical user interfacing. Many software companies publish GUI design style guides related to their particular commercial interfacing framework. Open Software Foundation's Motif (OSF/Motif) and Microsoft Windows are two products leading the way in setting industry norms in GUI style conventions. Much more general information can be found in a number of books.

A very straightforward, commonsensical, and yet, non-trivial exploration of the relevant issues can be found in Designing User Interfaces, by James E. Powell. Although this book

applies to graphical user interfaces as well, it covers user interfaces in general. Powell discusses such topics as user interfacing standards and human factors **[Powe90]**.

Screen design for human-computer interaction is very well documented and can be researched in such publications as "Human Engineering in Screen Design" by Galitz **[Gali83]**. Foley provides a very complete report on user-friendliness and human factors in computer graphics interaction **[Fole84]**. Screen design for iconic, multi-window interfaces is researched by Aaron **[Aaro84]**. Halter discusses interface design including feedback and message display in connection with a case study of several platforms **[Halt85]**. Meltzer discusses user-friendliness as the last frontier of (graphical) user interfacing **[Melt85]**. More technical information on problems in human-computer interaction and methods to improve it can be found in reference **[Fisc89]** and **[Wils91]** respectively.

Swezey and Spencer both provide sets of guidelines for interactive software design **[Swez83][Spen85]**. Myers explores techniques for the creation of interfaces in "User-Interface Tools: Introduction and Survey" **[Myer89]**. Lee describes tools for building interfaces in "User-Interface Development Tools" **[Lee90]**.

An object-oriented graphical user interface framework is described by Woyak. He explores the subject in great detail, developing classes for their eventual use in a graphical user interface for an application. Woyak explains the place of windows, menu items and event handling in an object-oriented scheme **[Woya92][Woya93]**. The framework and its use in this work will be described further in conjunction with the class development and class descriptions of the application.

The Object-Oriented Paradigm

Definition and Terms

In the object-oriented paradigm a problem domain is decomposed into entities characterized by having states, behaviors, and identities. These entities are referred to as **objects** or **instances**. The following terms are frequently used to describe facets of the object-oriented paradigm **[Booc91]**:

- ***class*** - A class is the type of a group of objects sharing the same data and methods. Unlike traditional types, a class can occupy a spot in a class hierarchy.
- ***encapsulation or information hiding*** - Encapsulation refers to the ability of an object to combine both data and methods within itself. In effect it thus hides the details of its structure and implementation. In C++ three levels of *visibility* allow for *private*, *protected* and *public* member data and functions. Private members can only be seen by the object which contains them. Protected members can also be seen by objects which inherit the object containing the members. Public members can be seen by anything with access to the containing object.

- ***inheritance*** - A class can be derived from a base class. It inherits all the data and functions from the base class and may add data and functions of its own in a step towards a more specialized class.
- ***polymorphism*** - Functions with the same name may be defined to have multiple actions depending on the types of the arguments sent to them. In C++ this is called *overloading* a function. Polymorphism also applies to class types such that objects of many different types, derived from a common base, may exhibit somewhat different implementations of the same member functions. In C++ such functions are declared as *virtual* functions in the base class.

There are basically two relationships between objects in an object-oriented system. They are often expressed as:

- ***"A" is a "B"*** - The "is a" relationship refers to inheritance. Since an object of class A inherits all of class B's data and methods it effectively is also an object of class B.
- ***"A" has a "C"*** - The "has a" relationship refers to the use of one object by another. If an object of class A encapsulates an object of class C as a member in its data it is said that the object of class A has an object of class C. If the object of class C is used only by A and not its other member objects, C is used only in *implementation*. If C also gives its other member objects access to A, then C uses A in its *interface*.

Advantages

Strict adherence to the object-oriented programming paradigm has been found to bring with it advantages over the procedural programming paradigm in the following areas:

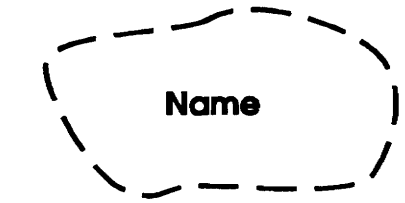
- ***extensibility*** - Encapsulation of data and methods, along with strong typing allows the creation of independent modules without the possibility of conflicts with existing objects. At the same time related and dependent objects may be created by taking advantage of the "is a" and "has a" relationships defined in the object-oriented paradigm.

- ***maintainability*** - Debugging of existing code is simplified by the isolation of errors to an object. Since all but the applicable parts of an object's implementation are hidden from the programmer he is only required to know and understand a small portion of the code at any given time to be able to repair it.

- ***re-usability*** - If object-oriented design is carefully, consistently and deliberately applied to the problem domain, a group of re-usable and flexible objects will result. They can be used in subsequent software, ideally without changes to the original objects. If an object is perfectly defined at the outset, it will contain all the data and methods that will ever be expected of it.

Notation

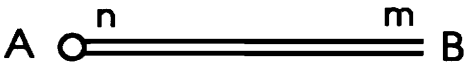
The notation shown below is used in all class diagrams in this thesis. It is used to describe inheritance and use relationships of classes and objects during the design of the class hierarchy [**Booc91**].



Class

Class "A" inherits from class "B"

"A" uses "B" in the implementation only



"A" uses "B" in the interface

The object "A" has n instances of the class "B".

The object "B" is had by m instances of the class "A".

The values of n and m can be

0	zero
1	one
*	zero or more
+	one or more
?	zero or one
n	the number value of n

The PHIGS Graphics Standard

Portability and machine independence of a piece of software can be ensured for the most part by adhering to strictly standard languages. Next to standard programming languages, the object-oriented graphical user interface (OO-GUI) requires a standard graphics language. The consistency provided also allows for integration of CAD design tools using this language.

While many graphics languages such as Silicon Graphics' GL or HOOPS are supported on different platforms, there are only two ISO (International Standards Organization) sanctioned standards. The Graphical Kernel System (GKS) became a two-dimensional graphics standard in 1985. The Programmer's Hierarchical Interactive Graphics System, PHIGS, ISO standard in 1988, improves on GKS to include structure editing features and three-dimensional graphics capabilities. Benefits of machine independence of (non-object-oriented) engineering applications written in PHIGS have been demonstrated in the ACSYNT conceptual aircraft design program **[Wamp88a][Wamp88b][Jaya92]** and other applications **[Jaya90][Jaya91][Jaya93a][Jaya93b][Schr91][Schr92]**. Examples of object-oriented applications of PHIGS are an object-oriented graphics system **[Wamp91]** and a graphical user interfacing framework emulating Motif **[Woya92]**. Both ACSYNT and the GUI framework will be referred to later in this thesis.

Graphical User Interfacing Framework

The graphical user interface framework developed by Woyak and used in this work provides certain classes as primitives. They can be combined as they are to make a graphical user interface, or they can be extended to create more specialized subclasses. A chart of the class organization, inheritance relationships and use relationships is provided in Figure 1.

The framework consists of five main classes. They are windows, interface managers, menu managers, menu items, and menu item managers. Menus, two and three dimensional views, and pop-up dialogue boxes are examples of windows. It is the interface manager's job to process all interactions between the user and the windows; it thus *has* or *uses* the window objects. A menu manager controls the operation of a group of menu items within a menu; the menu manager *has* the menu item objects. Menu items are akin to Motif Widgets and allow the user to interact with the menu. In cases where one menu item influences another menu item, a menu item manager is used to control their operation; the menu item manager *has* the interacting group of menu items. **[Woya92][Woya93]**

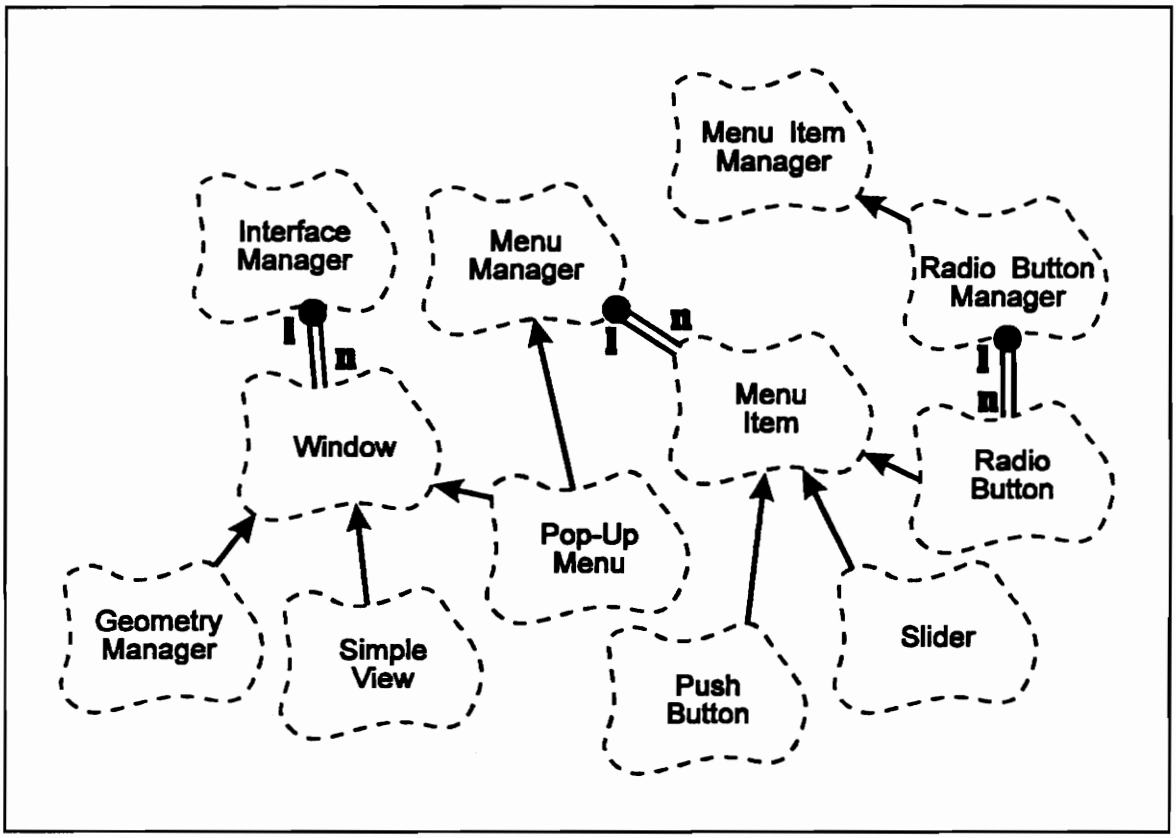


Figure 1 - Class Hierarchy of the Graphical User Interface Framework **[Woya93]**

Application Integration Options

Integration of two applications, such as ACSYNT and NEPP, can occur in any of four ways **[Penn92]**:

1. ***rigidly-connected interfacing*** - One application uses the other's output file for input. Since the data format is native to one of the applications, the other must have a translation routine to read the data.
2. ***rigidly-connected coupling*** - One application communicates with the other through program memory. One application must be adapted to be able to interpret the other's data structure and format.
3. ***freely-connected interfacing*** - One application uses the other's output file for input. The format used by both is a standard data exchange format such as IGES or PDES.
4. ***freely-connected coupling*** - One application communicates with the other through a third application managing incoming and outgoing requests for information from both applications.

NEPP - NASA Engine Performance Program

The NASA Engine Performance Program (NEPP) is a one-dimensional, steady-state thermodynamic engine performance cycle deck. Since its beginnings in 1975 as the Navy's NEPCOMP **[Cadd75]** it has evolved through NNEP **[Fish75]**, NNEPEQ **[Fish88]** and NNEP89 **[Plen91]** to the current NEPP. NEPP differs from the Navy/NASA Engine Program (NNEP89) only in that the Navy is no longer involved with the code. The program description and user's manual is still current in *The Navy/NASA Engine Program (NNEP89) -- A User's Manual.*

NEPP is capable of modeling almost any turbine engine, ranging from turboprop to air-turbo-rockets. This flexibility is achieved by combining a set of standard engine components in a user specified order while the program is executing. The list of available components includes:

- Inlet
- Duct
- Burner
- Water Injector
- Gas Generator
- Compressor
- Turbine
- Flow Splitter
- Ejector

- Heat-Exchanger
- Nozzle
- Shaft
- Load
- Propeller

Each component is defined by a list of up to 15 real numbers stored in a data file. Components are not connected directly to each other but rather through *flow stations*. *Flow stations* are non-physical entities used to describe the coupling between two or more components in terms of the flow and thermodynamic properties at the interface. The connection from a component to a flow station is specified by an additional array of four integers per component. Two of these integers are the numbers of the upstream flow stations and two are the numbers of downstream flow stations.

The only exceptions are the shaft, load and propeller. The shaft uses the four integers to specify the numbers of other components directly. Loads and propellers are specified as power drains or additions and cannot be connected to flow stations or components.

Control components are specified in a similar manner. Control components include:

- Variable Control
- Variable Optimization
- Variable Limiter
- Variable Scheduling
- Conditional Control

Several pre-processors, post-processors and user interfaces have been written for the various versions of NEPP. KONFIG and REKONFIG are two pre-processors written for NEPP in 1981 **[Fish81]**. These programs allow the user to input data line-by-line. A pre-processor written for use on VM/CMS systems in 1989 is the Simplified NEPP Automated Preprocessor (SNAP) **[Bert92]**. Several post-processors are also available.

The NASA Propulsion Analysis System (NPAS) is a very complete graphical user interface for NEPP. It includes graphical pre-processing and post-processing features. Being highly interactive it allows the user to configure his engine design by manipulating graphical components on the screen. Each engine component is defined using input menus in a list format. Controls are defined in the same manner. In addition to the regular NEPP thermodynamic analysis code it also supports the WATE code **[Onat79]** to calculate engine weight and flowpath dimensions.

The NPAS program makes use of the X Windows System and of the OSF/Motif graphical interface toolkit. Both of these are readily available for all UNIX workstations with the appropriate graphics hardware able to support an X server. The program is coded in ANSI C and FORTRAN 77. Since Motif is used for the graphics and some system calls are made it is not written completely in standard languages. Due to this and also due to the procedural design of the program it does not provide for easy integration with another CAD program.

ACSYNT - Aircraft Synthesis

The user interface for NEPP has been designed to be compatible with a future object-oriented NEPP and the object-oriented ACSYNT under development at the CAD laboratory. The eventual integration with the procedural aircraft conceptual design code ACSYNT has also been kept in mind in the design of the interface's classes.

ACSYNT, originally a batch-driven program, is the first modular conceptual design code for subsonic and supersonic military and commercial aircraft. Being developed since the 1970's at the NASA-Ames Research Center, it is divided into modules of subroutines handling the analysis for each of the different disciplines of aircraft design **[Mykl93]**. Work on a PHIGS based interactive CAD ACSYNT began in 1987 at the Computer Aided Design Laboratory at Virginia Tech **[Jaya92][Mykl93][Wamp88a][Wamp88b]**.

How does NEPP fit in with a procedural ACSYNT?

Both the original and the current graphical ACSYNT were programmed in the procedural paradigm. Figure 2 shows ACSYNT's modular program structure. It also clearly shows the program's function oriented approach. The new OO-GUI will be able to tie NEPP in with ACSYNT as an object-oriented engine class whose method is used to generate data requested from it by the propulsion module. Several other object-oriented modules have

already or are currently being integrated with ACSYNT in this manner **[Kell93][Rive93][Uhor93a][Uhor93b].**

Integration of the new NEPP interface into ACSYNT can occur in the propulsion module. The OO-GUI can be initiated from ACSYNT using a FORTRAN to C function call to initiate a C++ constructor. Since NEPP's potentially long execution time prohibits "on-the-fly" calculation of engine performance tables, they must be created and archived previous to running the aircraft convergence cycle. For this reason rigidly-connected interfacing can be used to communicate data from NEPP to ACSYNT. ACSYNT and NEPP both use versions of the format of the AMAC aircraft mission analysis program used at NASA - Lewis Research Center. Therefore translation routines already exist in both programs to convert the output/input to/from the AMAC "mediator" format to the applications' own format. ACSYNT does, however, require a specialized AMAC format such that some minor formatting and further integration work still has to be done on its end. To make ACSYNT and NEPP compatible on a functional level they had to be thoroughly checked for duplicate subroutine, function or common block names. To avoid conflicts in file input/output all of ACSYNT's file unit definitions were localized so they could easily be adapted to NEPP's file unit definitions.

The user interfaces of the two programs cannot be integrated as easily. Since both ACSYNT and the NEPP GUI use PHIGS, they both make use of the same graphics resources. To run both applications in the same window would mean saving the current state of the graphics resources and initializing them to the saved state of the new application each time the transition is made from one application to the other. An easier method involves the compromise of opening two workstations. In a windowing environment like X this has the effect of running ACSYNT in one window and the NEPP

GUI in another window. When the propulsion module is called the new window will open and control will be given to it until the module is exited and the window is closed.

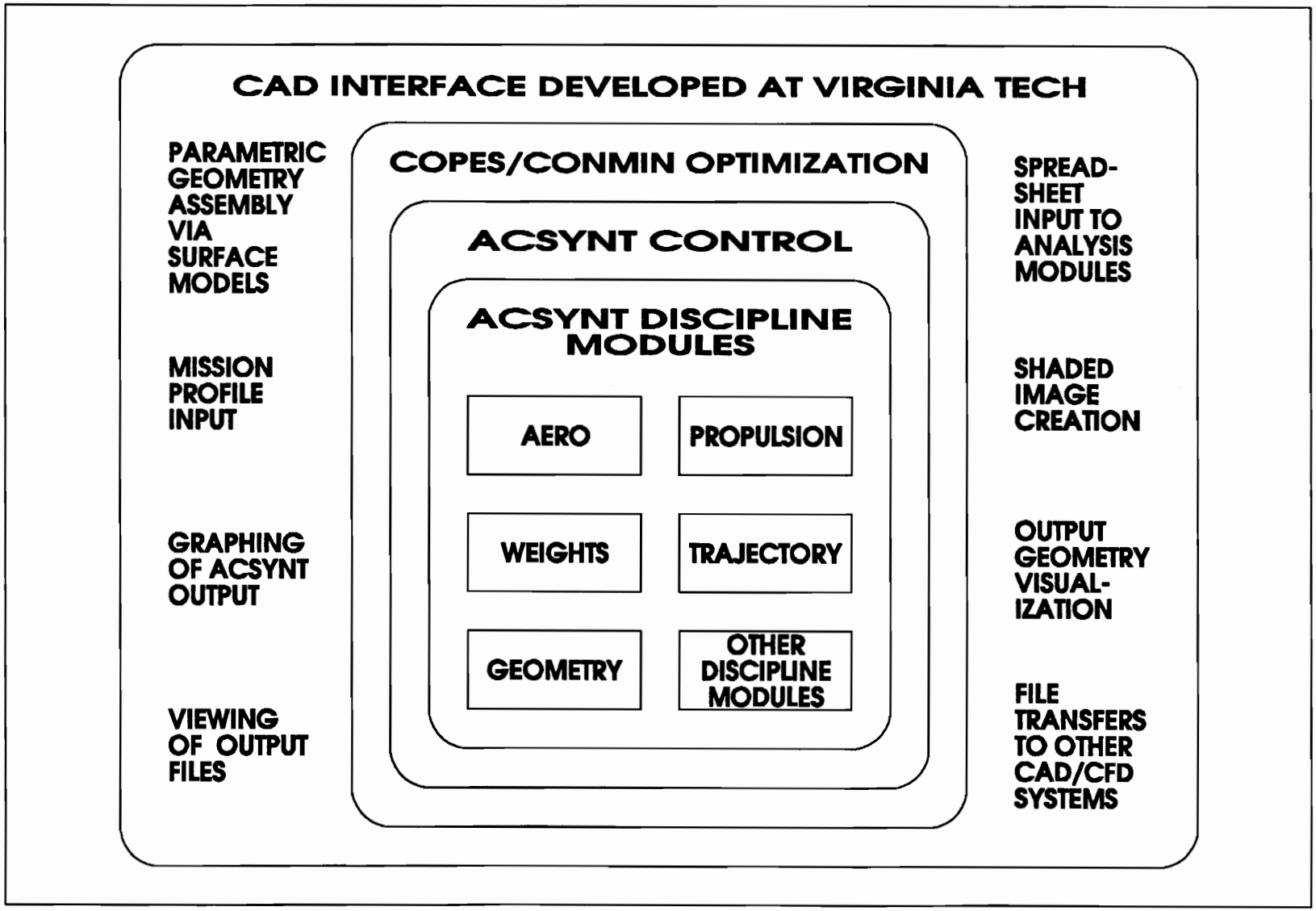


Figure 2 - Program Structure of Procedural ACSYNT(reconstructed from [Schr91])

How does NEPP fit in with an object-oriented ACSYNT

An object-oriented version of ACSYNT is currently being developed. Integration of an OO ACSYNT and the NEPP OO-GUI will be done by rigidly-connected coupling. Since the classes of the NEPP user interface and the object-oriented ACSYNT will be designed with an eye on each other's structure, the NEPP user interface will be able to supply information directly to the object-oriented ACSYNT through program memory. The class, Engine System, will be used in the decomposition of the aircraft into objects. The further decomposition and class descriptions are given below.

Object-Oriented Design of the Interface

Extensions to the Graphical User Interface Framework

The GUI framework provides the necessary foundation for building the utilities needed for the NEPP graphical user interface. Coded object-oriented in C++ and based on the PHIGS standard, the framework was developed by Woyak for extensibility and compatibility with other CAD applications. Thus, the five main classes of the existing framework were used and inherited, as part of this work, to create certain more complex and more specific classes for graphical user interfacing.

Scroll Bar

The scroll bar is a menu item, consisting of five other menu items. It has a slider, two push buttons and two arrows as member objects. The class diagram in figure 3 shows the relationships. The scroll bar can be used for specifying values just like a slider, but its strength lies with, as the name suggests, scrolling an image in a window.

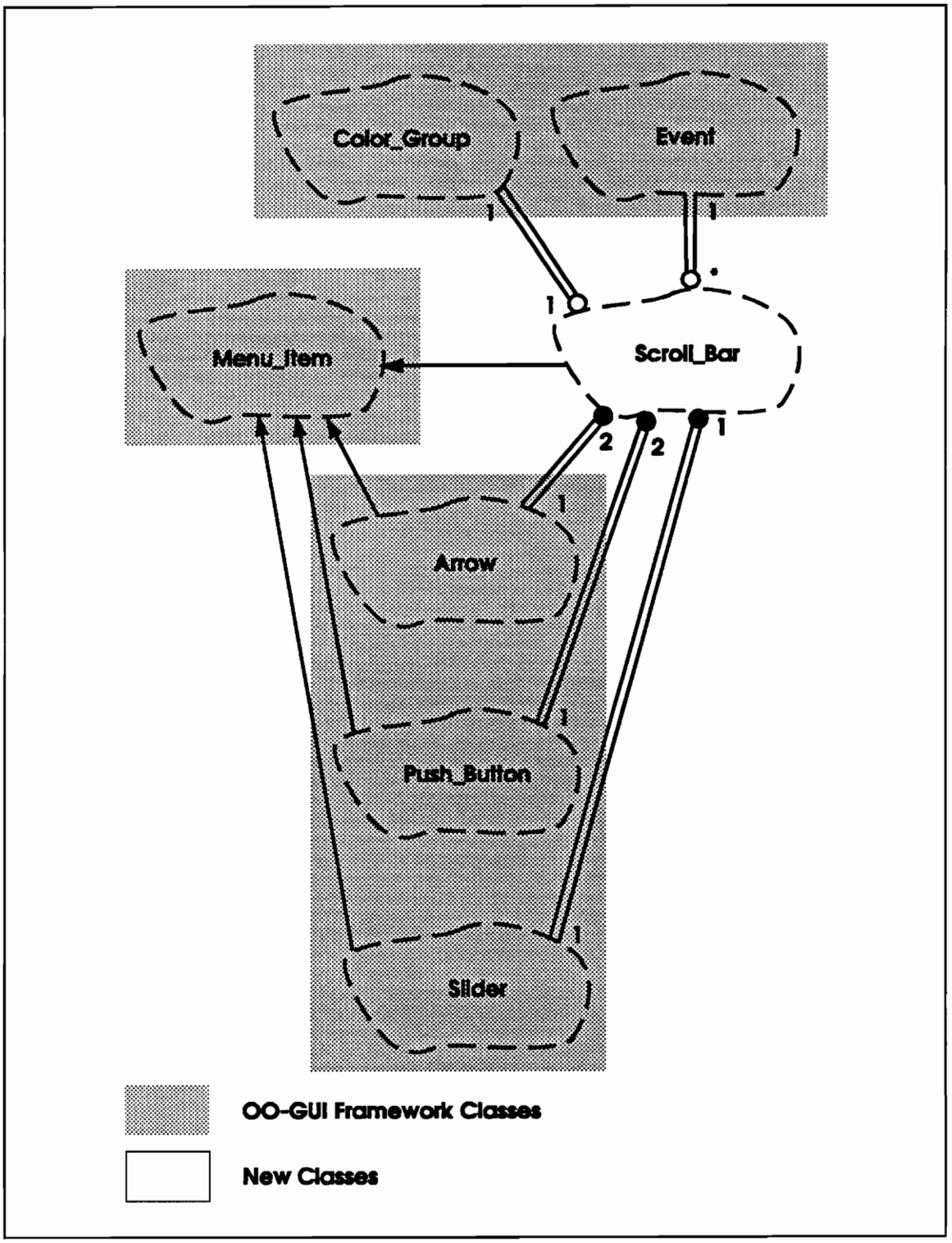


Figure 3 - Class Diagram for the Scroll Bar Class

Scroll Window

The scroll window is derived from the geometry manager base class. It has four member objects: two scroll bars and two push buttons. The scroll window class has all the behaviors of the geometry manager but adds the ability to size and move the displayed area in the geometry view through use of the scroll bars and push buttons. The class diagram is shown in figure 4 below.

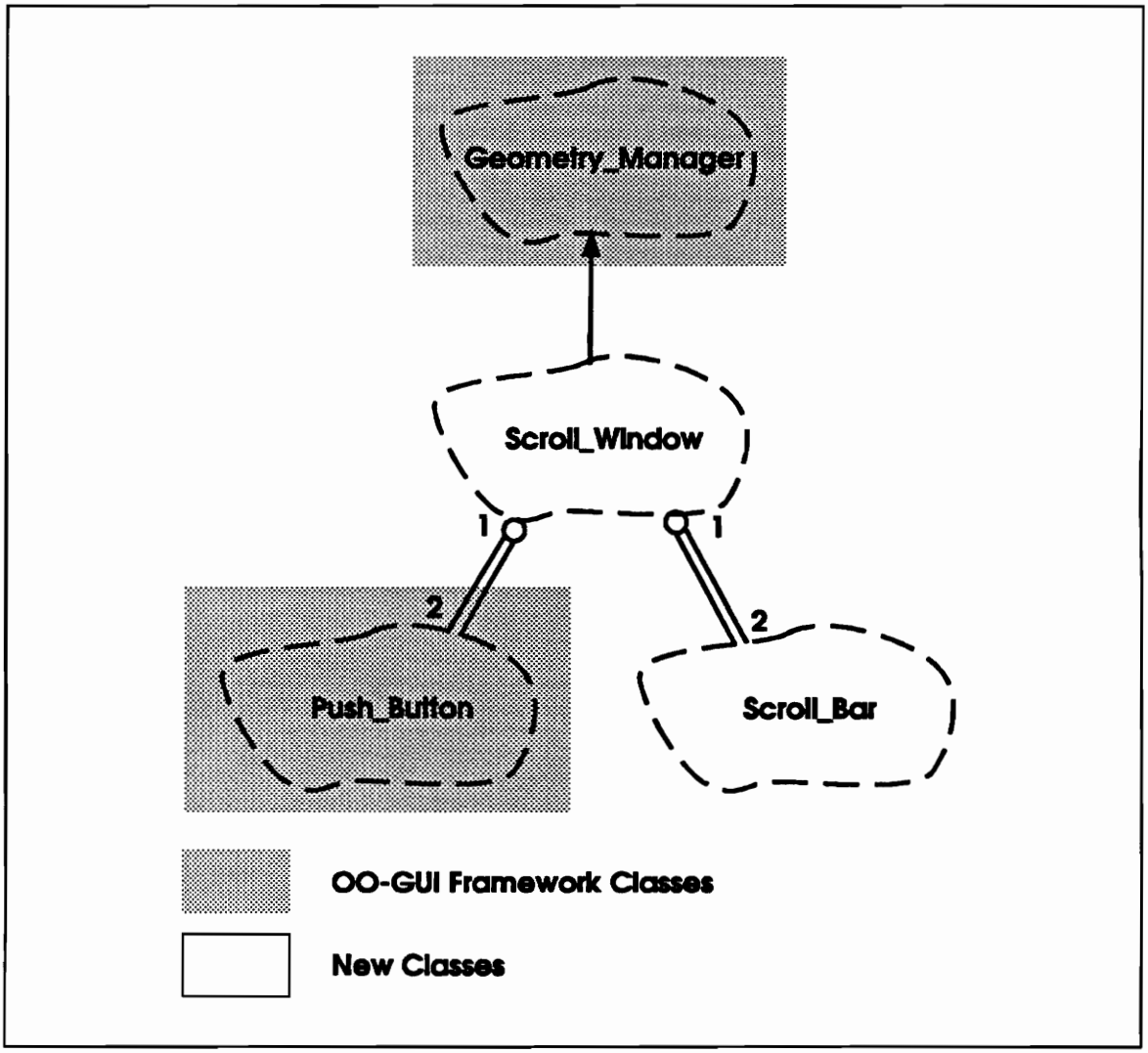


Figure 4 - Class Diagram for the Scroll Window Class

Formatted Pop Up Menu

The formatted pop-up menu is a base class. The class has the following member objects: one pop-up menu, label objects, push button objects, text input field objects and a matching number of frame objects. It also has a color group object for use by the menu items. The formatted pop-up-menu was created to facilitate the creation and editing of formatted input menus. The class diagram is shown in figure 5 below.

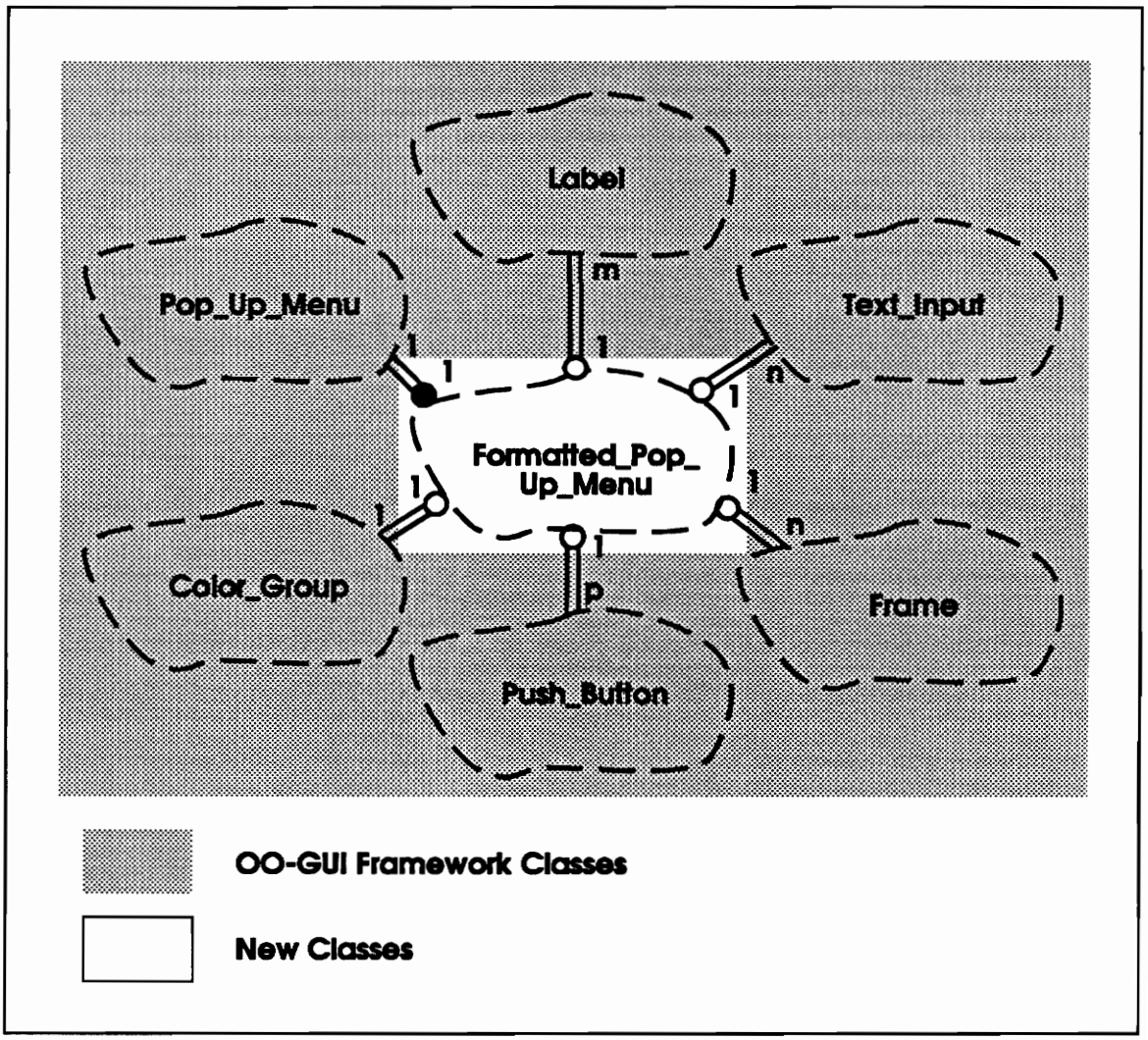


Figure 5 - Class Diagram for the Formatted Pop-Up Menu Class

Classes for the NEPP Graphical User Interface

The classes for the NEPP graphical user interface are, in part, designed as extensions of the GUI framework. Much of the class hierarchy, however, was determined by the decomposition of the engine configuration problem domain. Rather than building on graphical user interfacing classes, a new hierarchy was created which models the problem domain more naturally. These classes are provided with GUI facilities since that is one of their purposes.

Graphics Info

The graphics info class is a base class containing four objects. They are an interface manager object, geometry manager object, 3-D view object, and a color group object. The graphics info class allows all objects using it access to the data and methods of the afore-mentioned objects and some other relevant interfacing data. It is purely a utility class. The class diagram is shown in figure 6 below.

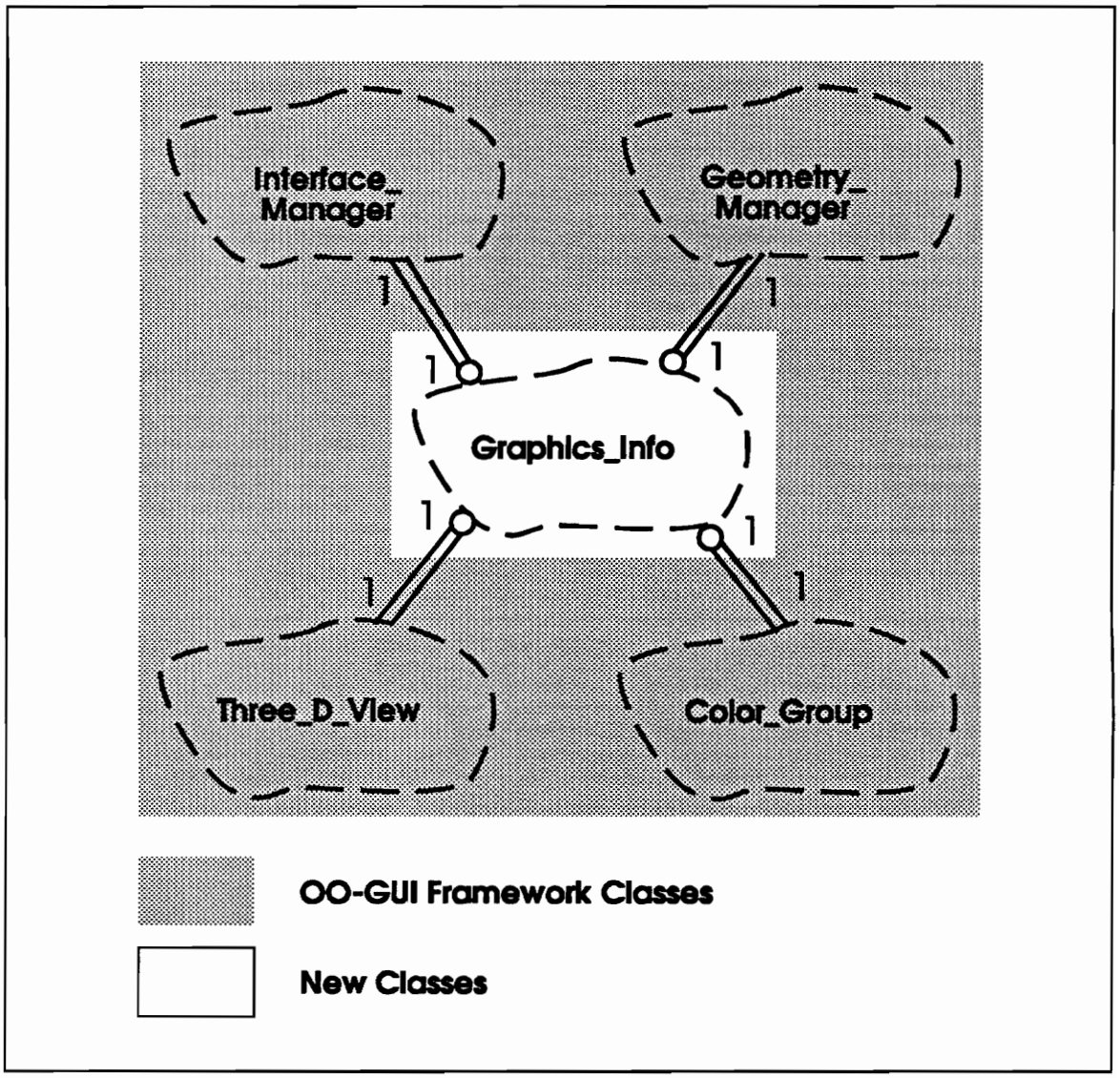


Figure 6 - Class Diagram for the Graphics Info Class

Configuration Window

The configuration window class inherits the scroll window class. It has three push button objects, an aircraft engine object and a graphics info object. The push buttons are used

only in the implementation of the class whereas the other two objects must also be visible in the interface. The configuration window provides the tools to create, load or modify an aircraft engine configuration and specification both graphically and through text input screens.

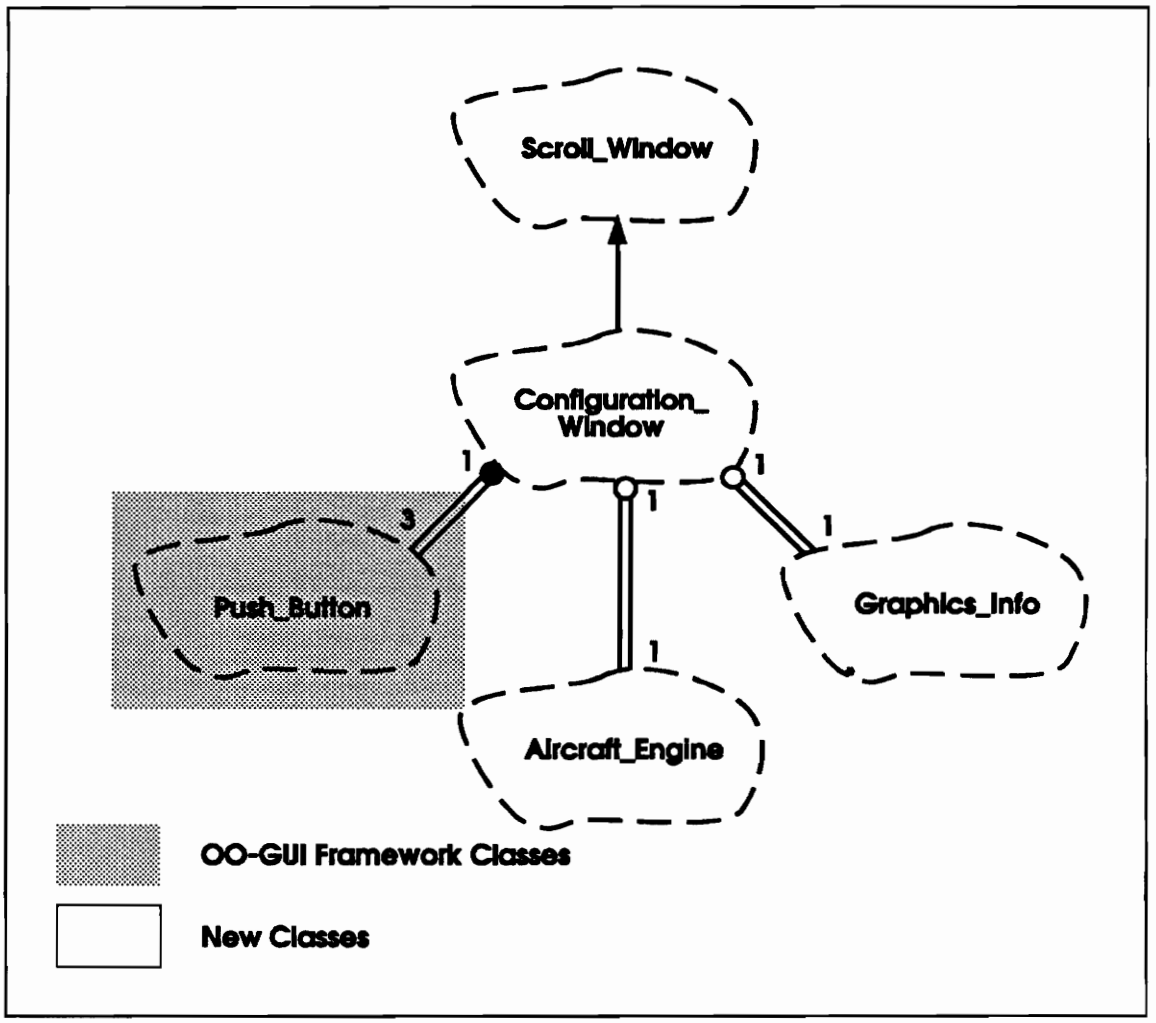


Figure 7 - Class Diagram for the Configuration Window Class

Aircraft Engine

The aircraft engine class is a base class. It contains the engine component objects and also has the connections between the components. This represents the physical configuration of the engine. The use of the connection objects is justified in the description of the connection class. The aircraft engine class diagram is shown in figure 8 below.

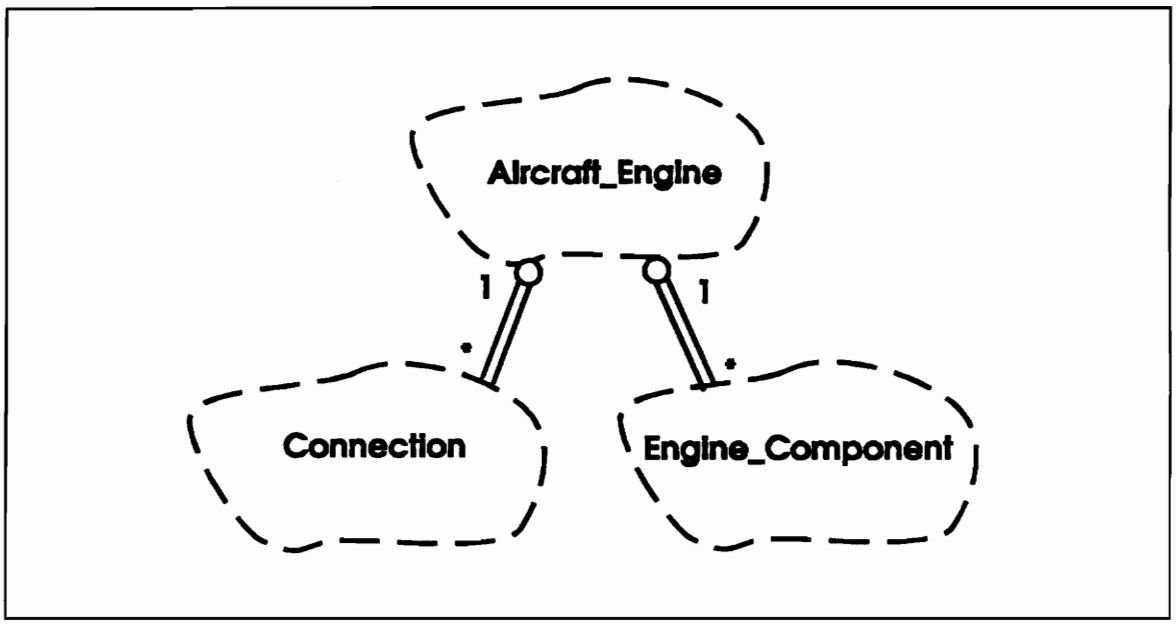


Figure 8 - Class Diagram for the Aircraft Engine Class

Engine Component and Derived Classes

The engine component is a base class. Its member objects are the graphics info object, an engine component icon and the connection objects associated with this engine configuration. Objects of this class are used to represent each physical component of the engine. They currently only contain information used by and provided for NEPP but can

be expanded to define other aspects of the engine component (geometry, weight, etc.). The class diagram is shown in figure 9 below

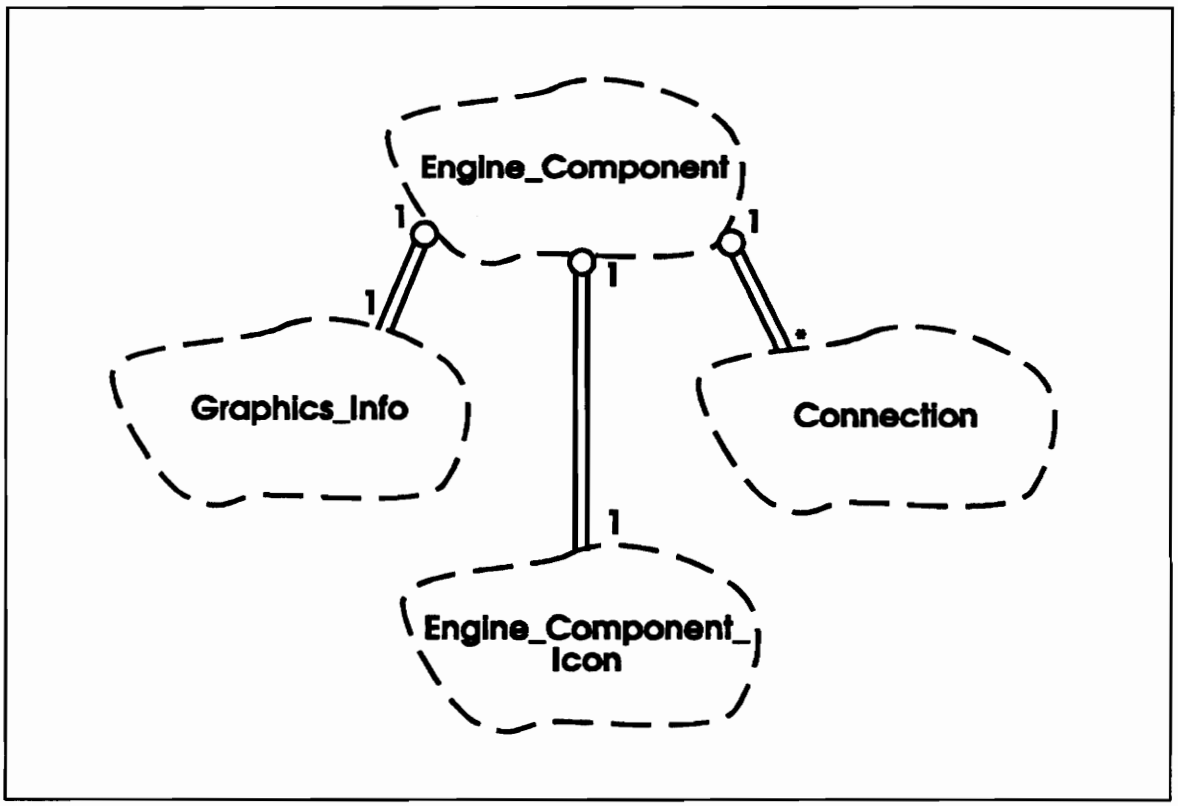


Figure 9 - Class Diagram for the Engine Component Class

Derived Engine Component Classes - First Iteration

The natural way to further decompose the engine component class is to break it apart into kinds of components. Some components are power sources, sinks or transfer components. Some components manipulate thermo- and fluiddynamic properties. The base engine component class should be broken up into at least these to categories. Components which

fit both descriptions can then inherit both classes. Others can only inherit one of them. The class diagram for this scheme is shown in figure 10.

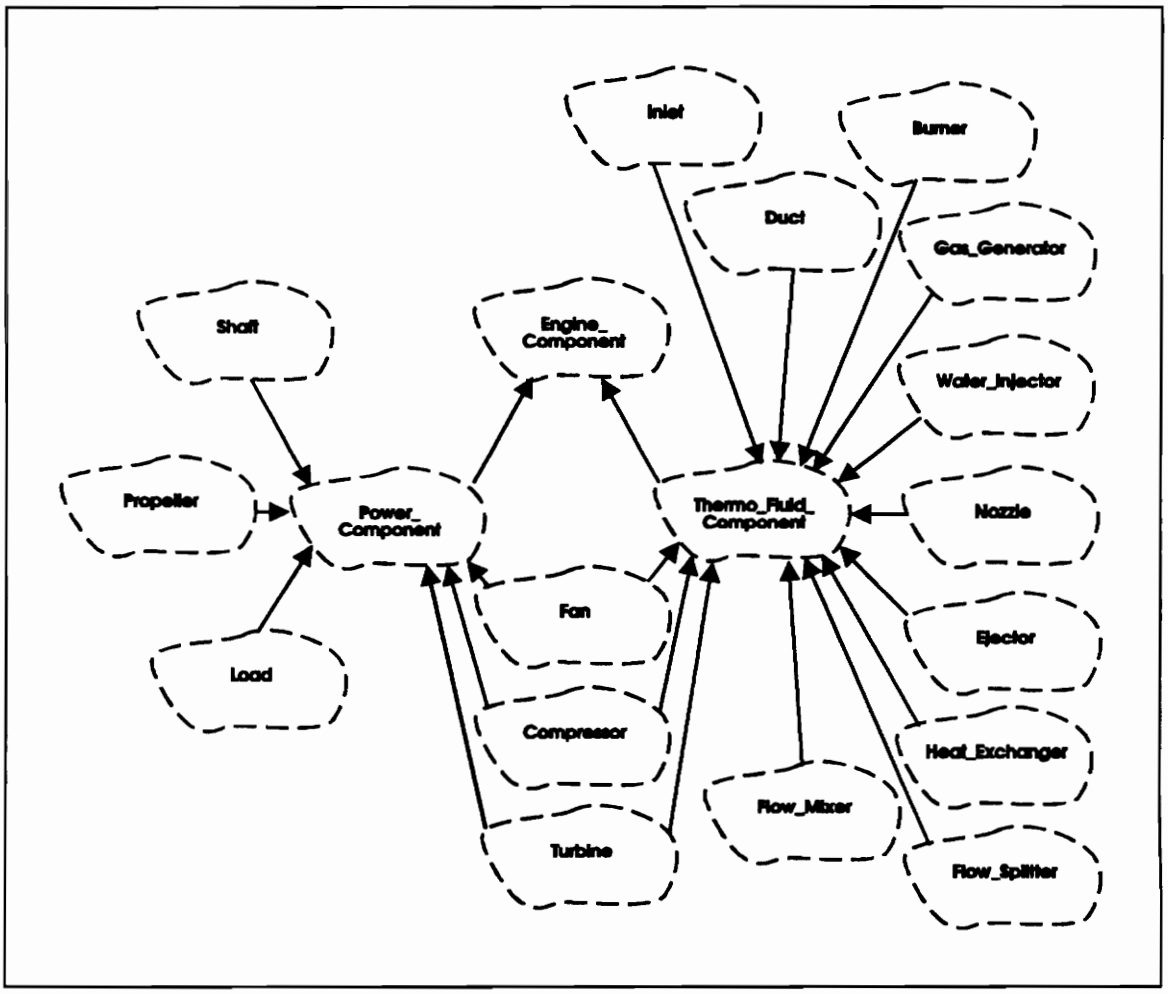


Figure 10 - Class Diagram for the Derived Engine Components - First Iteration

This class structure was not used. Multiple inheritance was not provided for in the first versions of C++. As a result, some compilers can only recognize a class which is derived from a single base class. The dangers of diamond-shaped inheritance, where a class is derived from two classes which are derived from the same base class, are stated comprehensively by Meyers **[Meye92]**. Due to the difficulties associated with diamond-

shaped inheritance, it is not allowed by some compilers. Figure 10 clearly shows the diamond shaped inheritance patterns formed by the turbine, fan and compressor class from both power and thermo-fluid component classes. The IBM C++ compiler, used exclusively in this project, recognized both power and thermo-fluid component as their base class, engine component. Thus a compile-time error appeared when, as the compiler saw it, the compressor class was derived twice from the engine component class.

Second Iteration

The first response solution to the diamond shaped inheritance problem was to create another class. Derived from the Engine_Component class it would carry specialized methods from both the power component class and the thermo-fluid component class. The thermo-fluid-power component class would serve as a base class to the components which would have been derived from both the power component and thermo-fluid component class in the first design. This arrangement, however, would render the power component and thermo-fluid component classes obsolete since all their characteristics and functionalities are embodied in the thermo-fluid-power component class.

The straightforward solution unfortunately falls prey to the tendency of the derived classes' methods to climb the hierarchy into the base class. This tendency is described by Booch and Meyers **[Booc91] [Meye92]**. For lack of any other solution, this is exactly what was implemented in this work. Rather than separating methods related to power components and methods related to thermo-fluid components into two classes, they are all coded in the base engine component class. In this arrangement there is only one level of

inheritance. All specialized components are derived directly from the engine component class. The final class structure is shown in Figure 11.

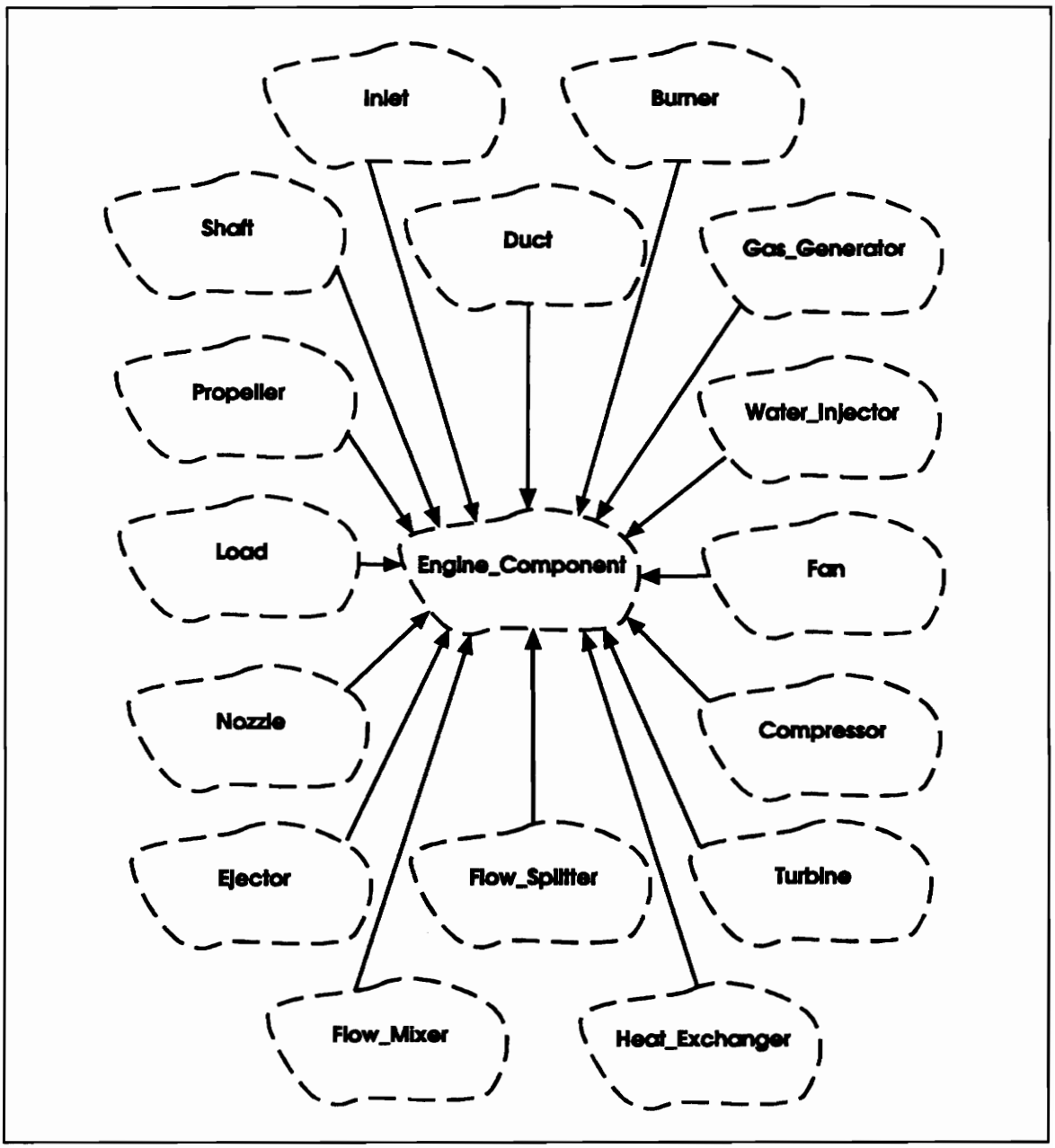


Figure 11 - Class Diagram for the Derived Engine Components - Second Iteration

Third Solution

A third solution, which was not pursued, involves the separation of each engine component into its flow and rotational constituents. In this case the class hierarchy, starting at Engine_Component, splits into Flow_Component and Rotational_Component. From this point the flow and rotational constituents of the individual engine components are derived from the Flow_Component and Rotational_Component. To join the constituents, they are used in their appropriate specialized engine component. This scheme is made clearer by the design diagram in figure 12. This solution avoids the diamond shaped inheritance and the shift of emphasis into the base class. At the same time, however, the individual engine components are separated from the base class. Thus, in this scheme, it cannot be said that a Compressor is an Engine_Component. This, therefore, may not optimally reflect the nature of the physical model.

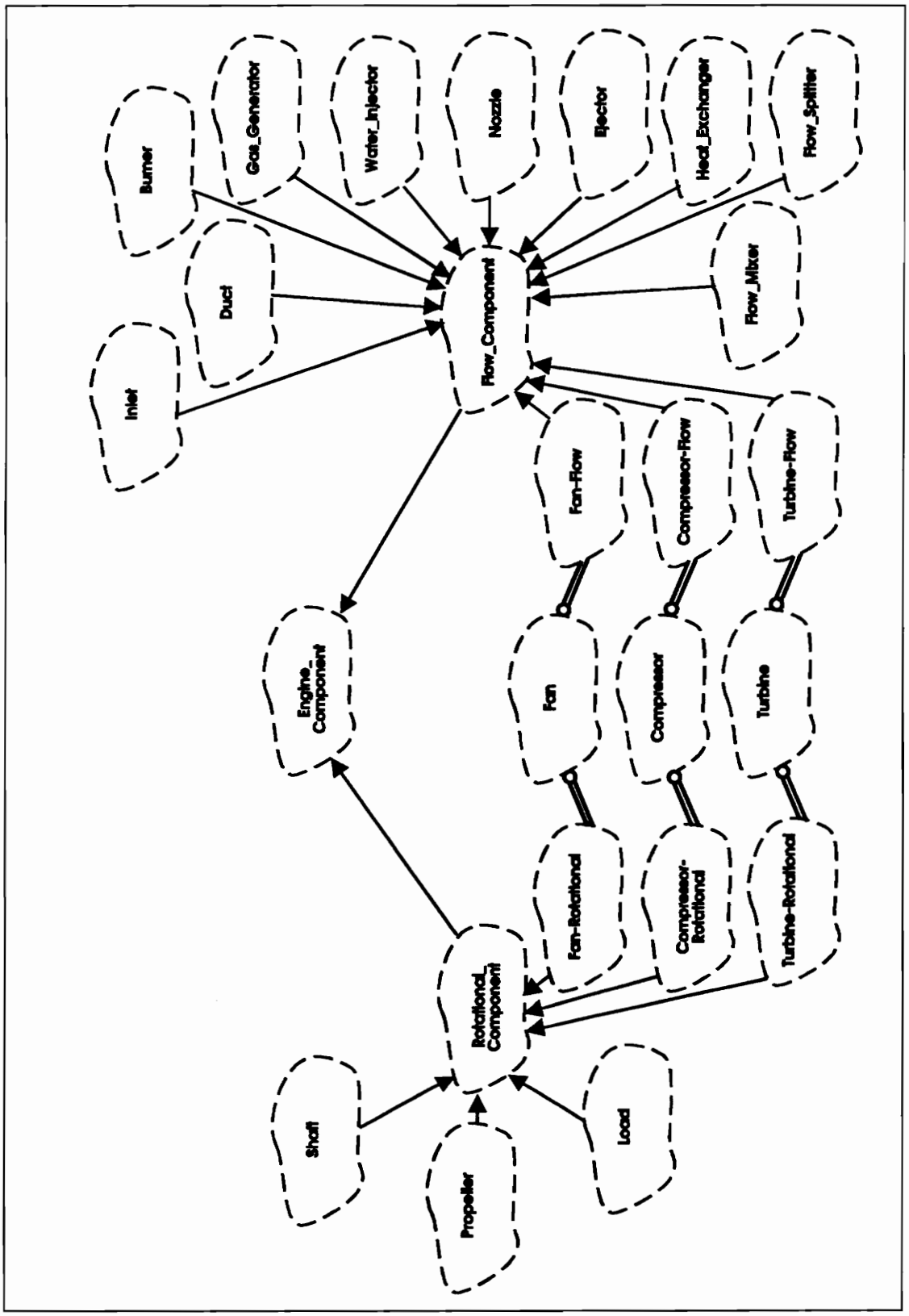


Figure 12 - Class Diagram for the Derived Engine Components - Third Solution

Connection and Derived Classes

The connection class is a base class using a graphics info object, an icon object and the engine component objects associated with this engine configuration. The justification for

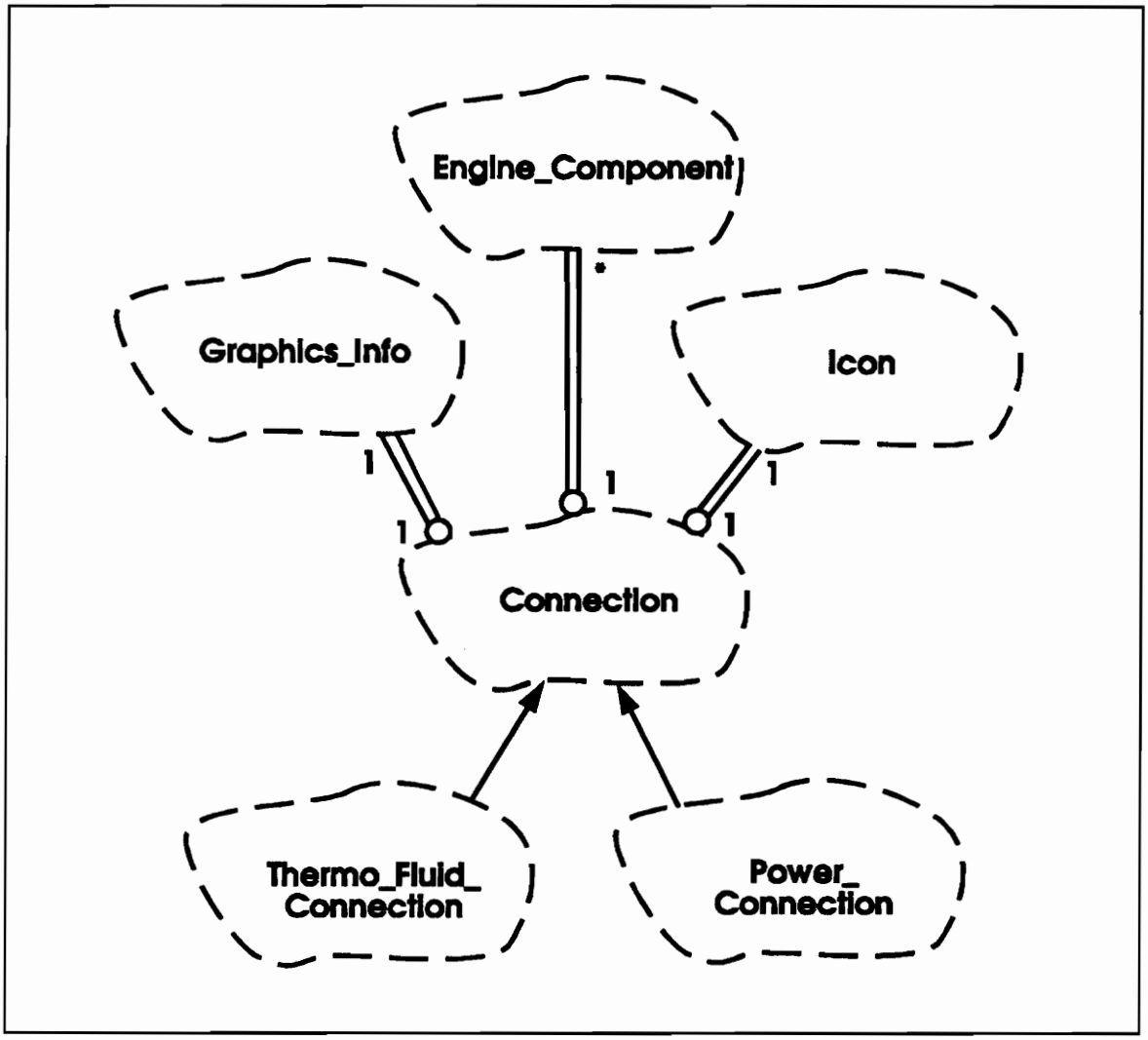


Figure 13 - Class Diagram for the Connection Class and its Derived Classes

the existence of this class is related to the mechanism of the NEPP flow station. The connections in a scheme without explicit connection objects can only model a physical interface between two components. It should not be necessary to connect more than two

components at the same physical interface since the Flow_Splitter and Mixer components are provided in NEPP to split the flow into two streams and to mix the flow from two streams back into one. Nevertheless NEPP has the built in functionality to do just that. In the traditional NEPP input lists, flow station numbers are used to specify connections between components. More than two components can specify a connection to the same flow station. In this way more than two components can be connected at the same physical interface. This functionality can only be accommodated by adding a class dedicated solely to the connection of components. The base class, connection, is inherited by two derived classes, power connection and thermo-fluid connection. The thermo-fluid connections are equivalent to the flow stations of the old NEPP. The class diagram for this arrangement is shown in figure 13 below.

Icon and Derived Classes

The icon class is a base class. Modeled after the push button class, it contains PHIGS structure identifier, PHIGS view, and color group objects. Unlike the push button class, it does not inherit the menu item class. It cannot, therefore, be managed. The icon class is the foundation for specialized icons to inherit. It provides general data and methods. The engine component icon, power connection icon, and thermo-fluid connection icon are three such specialized icons. The class diagram is shown in figure 14 below.

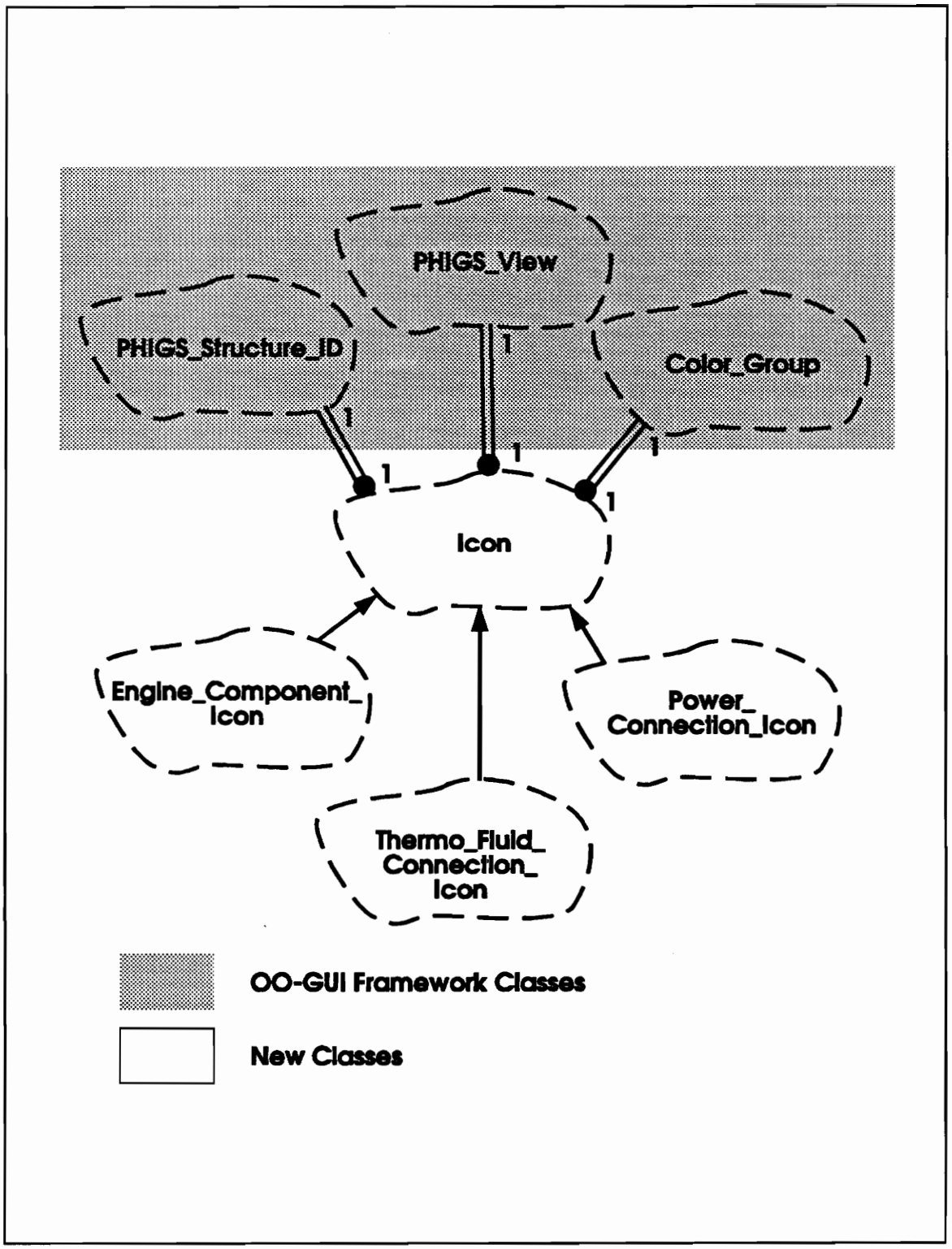


Figure 14 - Class Diagram for the Icon Class and Derived Classes

Class Descriptions

In this section the classes are explained by itemizing the characteristic data and methods. The descriptions at this point are still independent of programming language and thus, can be implemented in any object-oriented language.

Extensions to the Graphical User Interface Framework

Scroll Bar

The scroll bar is a combination of two push buttons, two arrows and a slider. By redefining their menu item processing functions, the behavior of the push buttons and of the slider can be modified to achieve the results required for a scroll bar. Like any menu item the scroll bar has position, and dimensions. In addition an orientation must be specified. The orientation can be either horizontal or vertical. The following functions characterize the behavior and functionality of the scroll bar:

- ***process_from_mouse*** - This is the event handler for mouse events. It intercepts and processes clicks and drags over the scroll bar menu item. The events recognized by the function include all of the events recognized by the

push buttons and the slider. If a push button was clicked, the value is incremented or decremented by *scroll_increment*. If the slider slot was clicked, the value is incremented or decremented by another user specified value, *page_scroll_increment*. If the slider was dragged, the value is changed by an amount proportional to the change in the slider position down to the smallest shift allowable by the resolution of the graphics display.

- ***get_percent_value*** - The scroll bar maintains a value from 0 to 100 depending on the current position of the slider within the menu item. This function returns this value.
- ***set_percent_value*** - The value and, with it, the position of the slider can be set from outside the scroll bar class using this function.
- ***set_scroll_increment*** - Set the increment by which the percent value is to be decremented or incremented when one of the push buttons is clicked.
- ***set_page_scroll_increment*** - Set the increment by which the percent value is to be decremented or incremented when the slider slot is clicked on either side of the slider.
- ***set_slider_length_percent*** - The length of the slider with respect to the length of the slot can be set using this function. For displays this can be used to let the slider represent the displayed portion of the total available information.
- ***set_change_event_id*** - The event identifier can be set using this function
- ***set_event_time_to_immediate*** - This function requests that events are to be responded to immediately.
- ***set_event_time_to_delayed*** - Specify that the value is to be updated when the menu manager's **ok_close** function closes the menu.

- ***set_color*** - The color combination of the scroll bar can be set using this function.

Scroll Window

The scroll window class inherits the geometry manager. The geometry manager is a scalable, sizable and movable menu. It can be raised and lowered with respect to other windows. It is made up of a title bar, a full-screen toggle and a three dimensional view encased by the scaling and sizing borders. It has provisions for displaying and processing more menu items which can be added by a programmer when a new class is derived from the geometry manager class.

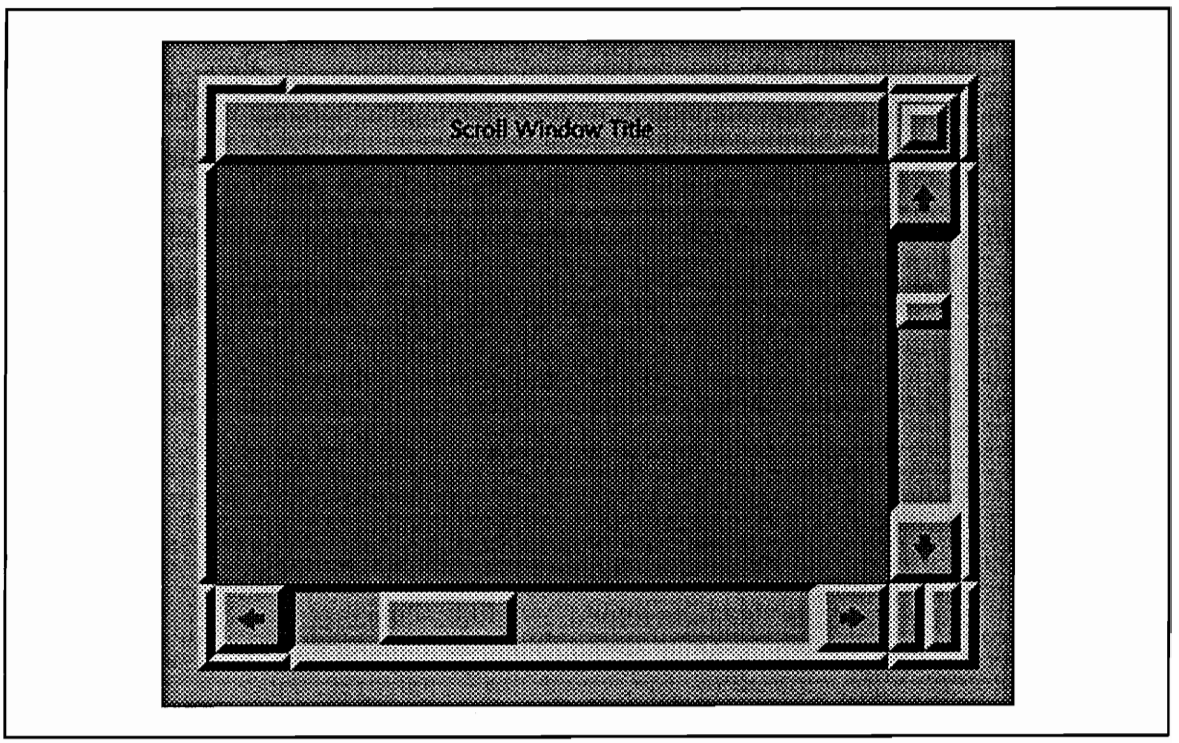


Figure 15 - Scroll Window

An object of the class scroll window has the general look and feel of a geometry manager except that provisions are made for horizontal and vertical scrolling and for zooming in and out of the displayed geometry. The width of the scroll bars is determined by the width of the window's title bar. The scroll window class can be inherited by further classes and more menu items can be added. This was done for the NEPP interface to create the main working area. A plain scroll window object is shown in Figure 15.

The following functions characterize the behavior and functionality of the scroll window:

- ***create_additional_components, delete_additional_components, process_additional_events, set_additional_differences*** - These functions allow the user of this class to add more menu items to the scroll window and to create room for displaying them in the window.
- ***process_geometry_view*** - This function is the event handler for events generated inside the geometry view.
- ***synchronize_scroll_window, synchronize_display_with_horizontal_scroll_bar, synchronize_display_with_vertical_scroll_bar, synchronize_horizontal_scroll_bar_with_display, synchronize_vertical_scroll_bar_with_display*** - Synchronizes the position of the display with respect to the position of the scroll bars or vice versa.
- ***calculate_page_scroll_percent_increments*** - Calculate the horizontal and vertical scroll percent increments corresponding to one display area.
- ***set_2D_3D_differences_rel*** - This function allows the placement and sizing of the geometry view with respect to the current geometry view.

- ***set_3D-3D-differences_abs*** - This function allows the placement and sizing of the geometry view with respect to the borders of the scroll window.
- ***get_horizontal_scroll_percent_value*** - Return the current percentage value of the horizontal scroll bar.
- ***get_vertical_scroll_percent_value*** - Return the current percentage value of the vertical scroll bar.
- ***get_horizontal_scroll_increment*** - Return the current scroll increment of the horizontal scroll bar.
- ***get_vertical_scroll_increment*** - Return the current scroll increment of the vertical scroll bar.
- ***get_horizontal_scroll_percent_increment*** - Return the current scroll increment in percent of the horizontal scroll bar.
- ***get_vertical_scroll_percent_increment*** - Return the current scroll increment in percent of the vertical scroll bar.
- ***get_model_width*** - Return the value currently stored as the width of the model.
- ***get_model_height*** - Return the value currently stored as the height of the model.
- ***get_zoom_factor*** - Return the value currently used to multiply or divide the magnification by when a zoom request is made.
- ***get_magnification*** - Return the current magnification.
- ***set_horizontal_scroll_percent_value*** - Manually sets the horizontal scroll bar percentage.
- ***set_vertical_scroll_percent_value*** - Manually sets the vertical scroll bar percentage.

- ***set_horizontal_scroll_increment*** - Set the horizontal scroll bar increment.
- ***set_vertical_scroll_increment*** - Set the vertical scroll bar increment.
- ***set_horizontal_scroll_percent_increment*** - Set the horizontal scroll bar increment as a percent value.
- ***set_vertical_scroll_percent_increment*** - Set the vertical scroll bar increment as a percent value.
- ***set_model_x_min*** - Set the minimum x-coordinate of the modeling space.
- ***set_model_y_min*** - Set the minimum y-coordinate of the modeling space.
- ***set_model_width*** - Set the width of the modeling space.
- ***set_model_height*** - Set the height of the modeling space.
- ***set_zoom_factor*** - Set the factor by which to zoom in or out on request.
- ***set_magnification*** - Manually set the magnification on the geometry view.

Formatted Pop Up Menu

This class is built around the pop-up menu primitive. (It does not inherit it!) It was created to facilitate the creation and editing of formatted input menus. The class accepts an array of text strings as arguments. It then parses these strings for format and escape sequences to automatically create a correctly sized menu with labels, text input fields and push buttons. The format strings are processed similar to the way the C library functions *printf* and *scanf* are processed. The following format and escape sequences are currently supported, where

f refers to a floating point number,

i refers to an integer, and

shadow thickness refers to the thickness of the beveled shadow on push buttons, frames, etc.:

line height:	**%*f*H**	If *f* is positive or zero the line height of the current and subsequent lines will be set to (*f* * character height). If *f* is negative the line height will be set automatically to the height of the tallest object on that line. Defaults: *f* = -1
line spacing:	**%*f*S**	The spacing following the current and subsequent lines will be set to (*f* * character height). Defaults: *f* = .5
"cursor" shift:	**%*f*>**	Shift internal cursor (*f* * shadow thickness) to the right before resuming drawing. Defaults: *f* = 1
"cursor" shift:	**%*f*<**	Shift internal cursor (*f* * shadow thickness) to left before resuming drawing. Defaults: *f* = 1
text input field:	**%*i*T**	A text input field with frame is constructed for a word with *i* letters. The height of the field is 2 * (character height + shadow thickness). Defaults: *i* must be specified.
Push button:	**%*i*B**	A push button is constructed. The push button is sized to fit a label with *i* characters. The height of the push button is 2

* (character height + shadow thickness).

Defaults: *i* must be specified.

Line feed: **/n** Inserts a line feed with carriage return using the currently set line height and spacing.

The following functions characterize the behavior and functionalities of this class:

- ***Formatted_Pop_Up_Menu*** - The constructor function creates the menu. This constructor takes care of initializing the class and calling the necessary functions to create the pop-up menu, create the menu items and finally add the menu items to the menu.
- ***create_pop_up_menu*** - This function is used to parse through the format lines to find the dimensions and positions of all the required menu items and of the menu's title. This is necessary to calculate the required width and height of the menu.
- ***initialize_line*** - Function used to process the format array. It initializes the parameters used to process one line and sets the line to be processed next.
- ***process_line*** - Function used to process the format array. This function is composed of calls to the following functions: **process_cursor_shift_right, process_cursor_shift_left, process_line_height_format, process_line_-spacing, process_text_input_format, process_button_format, process_-label**.
- ***terminate_line*** - Function used to process the format array. This function concludes the processing of a line.
- ***process_cursor_shift_right, process_cursor_shift_left, process_line_-height_format, process_line_spacing, process_text_input_format, process_-button_format, process_label*** - These functions parse the current line for a

particular format and calculate the location and dimensions for the corresponding menu item based on the position of the internal cursor and the current settings.

- ***create_menu_items*** - This function makes the actual menu items using the information obtained in the processing functions.
- ***add_menu_items*** - This function internally adds the menu items to the menu. This effectively tells the menu to manage and display the items.
- ***get_text_input_count*** - Retrieve the number of text input fields counted for this menu during processing.
- ***get_button_count*** - Retrieve the number of push buttons counted for this menu during processing.
- ***get_text_input_items*** - Retrieve the actual list of text input objects.
- ***get_button_items*** - Retrieve the actual list of push button objects.
- ***get_number_of_lines*** - Retrieve the number of lines counted in this menu during processing.
- ***get_text_input_id*** - Return the starting event id for the list of text input fields.
- ***get_button_id*** - Return the starting event id for the list of push buttons.

Classes for the NEPP Graphical User Interface

Graphics Info

This class collects all of the information necessary to create a graphically consistent object within the GUI. Some classes which do not necessarily have anything to do with graphics at all, may still have a need to communicate with the user. A large amount of information must be accessible to the class to accomplish this. This means one of two options: either the data is made globally accessible in scope or the data is passed to the class as an argument. The first option is considered unacceptable programming style for a number of well-known reasons. The second option means cluttering up the class with graphics information. To minimize the disorder, a class, Graphics_Info, was created to make GUI features available to otherwise no-GUI-related classes. Since the Configuration_Window class has access to all the relevant graphics information, it initializes and passes the Graphics_Info class to the appropriate member objects.

The base graphics info class is purely a utility class. It does not have methods of its own, but rather provides access to methods of its member objects. The following data is maintained by an object of the class Graphics_Info:

- ***iman*** - interface manager object
- ***window*** - geometry manager object
- ***geometry_view*** - view object used for geometry display
- ***character_height*** - text font height
- ***font*** - text font

- ***shadow_thickness*** - thickness of the bevel shadows (on push buttons, etc.)
- ***color*** - color object used to set menu colors

Configuration Window

The configuration window class is derived from the scroll window class. It is custom built to provide the ability to create, load or modify an aircraft engine configuration by using graphical and text input methods. The configuration window makes extensive use of graphical manipulation of engine components. This makes for a lot of flexibility but internally results in a complex class. A static menu of push buttons is part of the configuration window. They allow access to archiving features, window operation mode and provide a means to exit from the configuration window. The window operation mode can be toggled between automatically fitting the modeling space to the model size or fixing it. The following functions characterize the functionality and behavior of the configuration window class:

- ***set_additional_differences*** - This function redefines the corresponding function in the scroll window class. Room has to be made to fit the menu bar at the top of the window between the title bar and the display view. This function calculates the space needed and reduces the size of the display view accordingly. This function is called by the base class when the window is created.
- ***create_additional_components*** - This function redefines the corresponding function in the scroll window class. The additional menu items at the top of the

window are created in this function. This function is called by the base class when the window is created.

- ***delete_additional_components*** - This function redefines the corresponding function in the scroll window class. The additional menu items are deleted in this function when the destructor is called.
- ***process_additional_event*** - Any events generated by the additional components are processed in this event handler. This function is called from the base class event handler.
- ***pop_up_edit_menu*** - A pop-up menu can be activated to access editing options. The function first checks the current state of the configuration to decide which menu items should be deactivated and what the label for each menu item should say. For example if no components are highlighted the "delete" menu item is deactivated. If six components are highlighted then the menu item's label reads, "Delete 6 components". The function then creates the menu, manages it and enters an event loop. The loop is exited and the pop-up is deleted when a button is clicked.
- ***update_geometry*** - If the display mode is "Fit Geometry" then the display is updated and the function is exited; otherwise find the absolute minima and maxima coordinates of the model. Then information is relayed to the base class to induce it to update the size of the scroll bars and the position of the geometry. Lastly the display is updated.
- ***process_geometry_view*** - This is the main processing loop for the geometry display. Figure 15 explains the behavior of this function using pseudo code.

```
If nothing was picked (i.e. the mouse button was pressed over the blank background) then
    pop up the editing menu
Else if something was picked
    get the component or connection corresponding to the structure id
    If the left button was pressed then
        While the mouse button is not released
            Sample the location
            If the mouse was dragged then
                move the component
            If object was only clicked, not dragged then
                select-toggle the component ( this highlights the component )
    Else if the middle button was pressed then
        While the mouse button is not released
            Sample the location
            If the mouse is located over the icon then
                activate the icon
            Else
                deactivate the icon
        If the icon is active then
            Pop-up the input menu for the component or connection
            deactivate the icon
```

Figure 16 - process_geometry_view Pseudo Code

- ***add_component*** - This function adds an engine component to the linked list of displayed components.
- ***remove_components*** - Engine components can be removed using this function. When this function gets called, all selected (highlighted) components are removed. There is an option of executing this function with or without error checking.
- ***add_flow_connection*** - This function adds a flow station to the linked list of connections.
- ***add_rotational_connection*** - This function adds a rotational connection to the linked list of connections.

- ***remove_connections*** - Connections can be removed using this function. When this function gets called all selected (highlighted) connections are removed. There is an option of executing this function with or without error checking.
- ***connect_components*** - This function connects two components or a component and a flow station. It does preliminary error checking to make certain that the two items are compatible. The pseudo-code in figure 17 explains the procedure.

```
If there are not enough items to make a connection then
    display an error message and exit the function.
Loop while the user picks the two items to connect
    If the pick was in the background, exit the function
    else a component or a connection was picked...
        If the pick was a connection then
            set a flag to remember that it can only be connected upstream of a component
    If the second pick was in the background, exit the function
    else a component or connection was picked...
        If this item is the same component as the first, issue error message and exit.
        If the first item and this item are connections, issue error message and exit because
            two connections cannot be connected.
Connect the objects...
Figure out whether a rotational or flow connection is being done.  Then do preliminary
 error trapping to make sure each object allows a connection to the other.
To do a ROTATIONAL connection:
    If not both components are ROTATIONAL COMPONENTs
        issue an error message and exit the function
    else
        attempt to connect them ( Each component will do error checking and may
        still refuse to connect )
To do a FLOW connection:
    If the component is not a FLOW COMPONENT
        issue an error message and exit the function
    else
        attempt to connect the two objects ( The connection may still be refused by either object )
```

Figure 17 - connect_components Pseudo Code

- ***disconnect_components*** - This function disconnects two objects from each other. The function's error trapping makes sure that enough objects exist to

have a connection in the first place and that the two picked objects are connected. The objects are then disconnected using different methods depending on how they are connected.

- ***select_all*** - This function highlights all components and flow stations. All of the components and flow stations can then be edited simultaneously.
- ***unselect_all*** - All components and flow stations can be deselected simultaneously using this function.
- ***process_object*** - Process the formatted pop-up menu for object storage. The choices are "Engine...", "Engine Component..." and "Cancel". "Engine..." pops up the engine object storage menu. "Engine Component..." pops up the engine component storage menu.
- ***process_engine_object*** - Process the formatted pop-up menu for engine configuration storage. The choices are "Load", "Save" and "Cancel".
- ***process_component_object*** - Process the formatted pop-up menu for engine component storage. The choices are "Load", "Save", "Create" and "Cancel".
- ***read_NEPP_input_file*** - This function reads a NEPP input file for an engine configuration. It parses the file and executes all the steps automatically that would have been done manually to create the configuration.

Aircraft Engine

The aircraft engine class is a base class. It contains information regarding the engine configuration. It can also carry the state of the engine or geometry and weights information and methods. These, however, were not defined since this information is not

available from NEPP. The aircraft engine class currently only contains the list of connections and the list of components making for a complete configuration.

Engine Component and Derived Classes

The engine component class is the base class holding all the information and methods needed to describe an NEPP engine component. The class can be divided into data and methods in the following categories:

- linked list
- icon
- data input
- engine component configuration

The following functions are defined in the engine component class:

- ***move_selected_components*** - This function is responsible for moving the component icons on the display. It traverses the list of engine components and requests the move from each selected icon.
- ***count_selected_components*** - This function traverses the list of engine components and counts the components with icons in the selected state.
- ***get_component*** - This function is overloaded to either return a component requested by name or by number.
- ***PHIGS_get_component*** - This function returns the component with a specific PHIGS structure identifier.

- ***verify_saved_input*** - Makes sure all the connections specified and stored are still connected. If they are not, like for example in the event that a component or connection was deleted, the corresponding stored entry is blanked.
- ***process_push_buttons*** - This function handles push button events. In the engine component input menus push buttons usually have toggling labels. This function takes care of relabeling the buttons depending on the available choices for the particular component.
- ***verify_input*** - This function checks the current input against invalid values. Valid input are set and invalid inputs are set to valid defaults. The pseudo code in figure 18 explains the structure of this function.

```
Attempt to change the name of the component to the name in the input menu.
Attempt to connect the desired connections
    Loop through the rotational connection indices
        First make sure there is nothing connected in the current location
        If there is a rotational connection in the current location
            Store the name of the connected component
            Disconnect the two components by deleting the connection
        Get the name of the new component from the input field
        If the name is blank then set the return code to 0
        else find the corresponding component
        If the component is this component or NULL set the return code to REFUSED
        If the component is not a rotational component set the return code to REFUSED
        If the component is already connected through one of the rotational connections disconnect it.
        Create a rotational connection and associate both components with it.
            Create a linked list connection
            Set the name to the next available NEGATIVE integer in the list of connections
        If the new_connection is refused by either component
            disconnect any successful connection and
            reconnect the previously connected component if there was one.
    Loop through the desired upstream connection input fields
        If the desired connection is not allowed set the return code to REFUSED
    Loop through the desired downstream connection input fields
        If the desired connection is not allowed set the return code to REFUSED
Return the return code whether it be REFUSED or not
```

Figure 18 - verify_input Pseudo Code

- ***set_type*** - This function is only allowed in the base class since the specialized component classes cannot change their type. In the base class this function is overloaded to either pop up the type input menu or set the type directly.
- ***pop_up_type_menu*** - This function pops up and processes a menu allowing the user to choose a component type from a list of radio buttons labeled with each component type.
- ***pop_input_data_menu*** - This function creates the input menu according to the format array specified for the component.
- ***process_input_data_menu*** - Processes the input pop-up menu. The push buttons receive special attention since they must be toggled by the component. The input field entries are only checked for validity when the "Ok" button is clicked. While invalid field entries are encountered the menu cannot be exited with the "Ok" button. The "Cancel" button exits without further processing.
- ***set_defaults*** - Read defaults into the input data.
- ***change_name*** - This function sets the name after error trapping. The error trapping ensures that the new name is a valid name and does not already exist.
- ***update_input_data_menu*** - This function updates the text input fields and button labels according to data stored in the component. This function may be used to reset the input menu to a previous state.
- ***set_name*** - This function is overloaded to either pop up the input menu and set the name input there or to set the name given as an argument.
- ***set_number*** - This function sets the component number used by NEPP.
- ***get_name*** - Return the component's name.
- ***get_type*** - Return the component type.
- ***get_number*** - Return the component number used by NEPP.

- ***can_connect_upstream, can_connect_downstream, can_connect_rotational*** - These functions do preliminary error trapping to determine whether this component is allowed to connect to a specified connection.
- ***connect_upstream, connect_downstream, connect_rotational*** - These functions are overloaded. The one set checks all conditions to check if the connection can be allowed. The other set is more aggressive and creates room for itself using the disconnect function to replace a previous connection. The outcome of both sets of connection requests depends on the result of the error trapping built into the base class and the derived classes.
- ***is_connected_to*** - Checks if this component is connected to another specified component via upstream or downstream flow station or rotational connection.
- ***count_local_connections*** - Counts the number of local upstream, downstream and rotational connections.
- ***disconnect*** - This function is overloaded to either disconnect from a rotational component by deleting the rotational connection or to undo a specified connection.
- ***disconnect_all*** - This function frees the component from any connection.

Connection and Derived Classes

Connections between two components, whether they be rotational or flow components, use object of the class type Connection. There is an ambiguity in the way the term connection is used in this work. A connection can refer to the Connection object as the link between two engine components. This kind of connection is visible to the user in the

flow station icons and their associated arrows and in the lines indicating a connection between two rotational components in the configuration. It can also refer to the internal link between the Connection object and the Engine_Component object. This kind of connection is visible to the end user only in the arrows between a flow station and flow component.

As already mentioned there are differences in the way a flow connection ("flow connection" and "flow station" are used interchangeably) and rotational connection behave and look. They do, however, have a fundamental resemblance. For this reason two classes, Flow_Connection and Rotational_Connection, were derived from the Connection base class. The base class contains everything common to both derived classes: linked list methods, some of the generic methods required for connecting the Connection objects to engine components and icon objects and methods for moving them.

The characteristic connection related functions are inquiry and virtual functions:

- ***get_connection*** - This function is overloaded to return the Connection object corresponding to a specified number or a character string name. This is done by traversing the linked list of connections until the correct object is found.
- ***PHIGS_get_connection*** - This function does the same thing as **get_connection** by matching the PHIGS structure id of the icon to a specified id. This function could not be overloaded because, while the PHIGS structure id argument and the number argument are of different types (**Pint** and **int**), they were ultimately both recognized as **int**'s by the compiler.
- ***is_connected_to, count_local_components*** and ***disconnect*** - These virtual functions need to be redefined in each derived class. The function **is_connected_to** checks if a component is connected to this connection. The

function **count_local_components** counts the connected components by counting the pointers to component names in the local name pointer arrays. The connection functions in the derived classes, Rotational_Connection and Flow_Connection, mirror the ones used in the Engine_Component class.

Icon and Derived Classes

The Icon class was created to display objects in the 3D geometry view. Some internal similarities to menu items exist. The icon object, however, is not managed as the menu item is, but rather controlled directly through an event handler. The Icon class is very flexible to allow a lot of freedom when creating the form and function of the derived icon class. The Icon class is the base class for the derived classes, Rotational_Connection_Icon, Flow_Connection_Icon and Engine_Component_Icon.

The following functions are the functions defined in the Icon base class:

- ***create_structure*** - This virtual function must be redefined in the derived class to create the PHIGS graphics structure.
- ***display_change*** - This function displays change in the icon's shape, size and position.
- ***associate*** - This function associates the icon structure with the specified PHIGS view.
- ***disassociate*** - This function frees the icon from the PHIGS view.
- ***get_structure_id*** - This function returns PHIGS structure id.

- ***get_state*** - This function returns the state of the icon. It can be toggled between **ACTIVE** and **INACTIVE.**
- ***get_select_state*** - This function returns the selected state of the icon. It can be toggled between **SELECTED** and **UNSELECTED.**
- ***get_width, get_height*** - These functions return the dimensions of the icon in modeling coordinates.
- ***get_x_ptr, get_y_ptr, get_width_ptr, get_height_ptr*** - The dimensions and coordinates of the icon can be monitored from outside the class, without having direct access to it, by using the address of the variable. These functions return that pointer.
- ***get_x_min, get_y_min, get_x_max, get_y_max*** - These functions return the minimum and maximum coordinates of the icon.
- ***get_x_center, get_y_center*** - These functions return the center coordinates of the icon.
- ***activate*** - This function changes the state of the icon to **ACTIVE.**
- ***deactivate*** - This function changes the state of the icon to **INACTIVE**
- ***select*** - This function changes the select state of the icon to **SELECTED**
- ***unselect*** - This function changes the select state of the icon to **UNSELECTED**
- ***select_toggle*** - This function toggles the select state of the icon between **SELECTED** and **UNSELECTED**
- ***set_color*** - This function sets the color of the icon.
- ***move_to, move_to_x, move_to_y, move_delta, move_delta_x, move_delta_y*** - These functions allow the icon to be translated in absolute or relative terms.
- ***set_name*** - This function sets the name of the icon.

The Engine_Component_Icon distinguishes itself from the base class by adding the four internal functions:

- ***create_name, create_lower_right_bevel, create_upper_left_bevel, create_face*** - These functions are called by the **create_structure** function to make the icon graphics. The engine component icon takes the shape of a push button sized to accommodate a name label specified by the user.

The Rotational_Connection_Icon adds the functions:

- ***create_line*** - This function is called by the create_structure function to make the icon graphics. The rotational connection icon takes the shape of lines connecting an array of anchor points.
- ***add_line, remove_line*** - These functions add and remove lines from the icon by adding or deleting anchor points.

The Flow_Connection_Icon adds the functions:

- ***create_name, create_lower_right_bevel, create_upper_left_bevel, create_face, create_arrow*** - These functions are called by the **create_structure** function to make the icon graphics. The engine component icon takes the shape of a push button sized to accommodate a name label specified by the user with incoming and outgoing arrows to specified anchor points.
- ***add_upstream_arrow, add_downstream_arrow, remove_arrow*** - These functions add and remove arrows from the icon by adding or deleting anchor points.

Notes on the Implementation

Tools Used

All development and testing of the graphical user interface was done on IBM RISC System/6000 workstations. Some integration coding was verified on a Silicon Graphics IRIS 4D workstation. Workstations in this range are becoming the platforms of choice for the development of large engineering applications because of their speed and graphics capabilities.

The operating system on both machines was UNIX. More specifically IBM AIX Version 3 and Version 3.2 and Silicon Graphics IRIX were used. UNIX provided the flexibility and utilities which are necessary for an efficient and pleasant to use software development environment.

Three programming languages were used. Both ACSYNT and NEPP are coded in part or entirely in FORTRAN 77. Integration of NEPP with the GUI and preparing ACSYNT and NEPP for integration required some coding in FORTRAN 77.

C code was used as an interface between FORTRAN and C++. Since C is a subset of C++ this differentiation is almost moot. As discussed earlier, however, the connection between

C and FORTRAN is easier to establish than a connection between FORTRAN and pure object-oriented C++.

C++ was chosen for the development of the OO-GUI. C++ exhibits all the traits necessary for programming in the object-oriented paradigm. Since C++ is a derivative of the popular C programming language, it is also rapidly becoming a favorite for object-oriented software projects. C++ compilers are now available for most workstations. Some compilers actually act as translators by first converting the C++ code into C code.

The object-oriented graphical user interface is built upon C++ classes provided in the PHIGS-based Motif-like object-oriented interface framework described by Woyak **[Woya92]**. The framework provides classes for such GUI necessities as windows, menus, buttons, sliders, text input and event handling.

PHIGS was chosen as the graphics language. PHIGS is the ISO standard for three-dimensional graphics and is available on a wide variety of platforms. The use of PHIGS in the GUI helps ensure device independence. The coding of the first (procedural) prototype was done using graPHIGS, the IBM implementation syntax of PHIGS.

The Annotated C++ Reference Manual **[Elli90]** and Borland's Turbo C++ library were consulted for C++ usage, syntax, methods and quick reference of standard function library entries. Meyers' Effective C++ was used to forestall and recover from some of the pitfalls of C++ programming **[Meye92]**.

Formatted Pop Up Menu

The following format string array, along with default button labels and text input values, produced the inlet data input menu shown in figure 22:

```
%1.0S%-1.0HInlet name              %20T
%0.5S%-1.0HUpstream connection     %3T
%2.0S%-1.0HDownstream connection %3T
%0.5S%-1.0H%32B  = %7T
%-.5S%+0.0H                                                    %>o
%0.5S%-1.0HFree stream temperature          %3>    = %7T    R
%0.5S%-1.0HFree stream static pressure      %3>    = %7T psi
%0.5S%-1.0HMach number at entrance          %3>    = %7T
%2.0S%-1.0HAltitude measured as %12B = %10T ft
%0.0S%-1.0H Additive drag
%0.0S%+0.5H----------------  = %7T psi
%0.5S%-1.0HDynamic pressure
%2.0S%-1.0HPressure recovery = %7T
%1.0S%-1.0H             %6B              %6B
```

Programming the data input menus using the Formatted_Pop_Up_Menu class allows comparatively easy additions or changes. In the case where the format string arrays are read in from files rather than hard-coded, the look of the menu may even be edited at run-time. This means that many different variations can easily be considered to get the desired layout. One drawback of using files to store menu format information is that the end user could conceivably tamper with the menu layout. Another drawback is the added delay in popping up a menu due to the slowness of the file system compared to hard coding. The string arrays clearly ought to be incorporated into the code before release to the end user.

Since the processing loops for many menus are very dependent on the purpose of the menus no attempt was made to process the formatted menus from within the class. Instead, all of the necessary information about the menu items can be accessed through member functions. Once this information is known, a customized processing loop can then be created.

Classes for the NEPP Graphical User Interface

Configuration_Window

The Configuration_Window class is the main driver for the NEPP OO-GUI. A configuration window provides the work area for graphically designing the engine. Effectively containing all of the GUI for engine development in one class makes the program easy to integrate into another object-oriented code.

The window has a menu bar across its top for accessing stored engine or component designs and saving new designs. The menu bar also provides an exit button and an option toggle between working with a fixed screen and working with an automatically centered and fitted screen. This option was necessary to avoid the "jumping" of the work area everytime the overall dimensions of the engine graphic was modified and the centering was adjusted. The Configuration_Window class contains all the processing functions for the menus and methods for calculating the automatic centering.

Engine_Component Class and its Derivatives

The set of engine components is implemented through the use of a doubly linked list. This differs from the NEPP implementation, where an array is used. The Aircraft Engine class maintains a pointer to an element of the doubly linked list of engine components.

The linked list function set includes these relatively standard methods:

count_elements

get_first

get_last

get_previous

get_next

add_element

insert_element

remove_this_element

bypass_this_element

The same methods are used in the Connection class's doubly linked list as well. The use of a class template could have saved the trouble of recoding these functions for the Connection class. Templates, however, are not consistently implemented on all C++ compilers at this time.

Many of the Engine_Component class's methods and data are programmed in such a way as to be independent of component type. Where this was not possible the function or data is redefined in the specialized derived class.

Data input occurs through the use of the formatted pop-up menu. The Engine_Component class contains the generic processing, error trapping and default setting functions used by all of the specific component types. Each specific component type class has error trapping functions specialized for the individual input data type. Text input error trapping checks if the input field is a valid number or special word and within the maximum and minimum values allowed for that input. Special words include "Map" for some component inputs where off-design performance maps are valid input sources. This feature was installed to prepare for the eventual addition of off-design input specifications.

The base class, Engine_Component, contains all of the functions related to connecting it to flow stations or to rotational components. In this area a lot of methods were programmed twice, because of the inclusion of both rotational and flow component methods in the base class. Connection requests are filtered through a sets of connecting functions in the base class, derived classes and the Connection classes. The outcome of connection requests depends on the result of the error trapping built into the base class and into the specialized component classes.

Connection Class and its Derivatives

Since information in flow stations should be accessible to the user, the Flow_Connection class has a pop up menu and the processing functions to go along with it. It displays the names of upstream and downstream connected components and allows the user to change the assigned name of the flow station to any positive integer. The class also contains the specific methods for connecting flow components upstream and downstream.

The Rotational_Connection class does not have a display menu since no information associated with it besides the actual connection is of value to the end user. This class, like the Flow_Connection class, contains specific methods for connecting to rotational components. The names of rotational connections are also integers. They are, however, negative non-zero numbers.

The movement and shape of connection icons is dependent on the location and movement of the connected components. Each connection object maintains a list of pointers to the coordinates of the connected components. In this way the coordinates used by the connection object to calculate the arrows are always up to date, even if the components move. The buttons of the flow connections can only be moved by being dragged with the mouse.

The list of names of components linked to a connection object is stored in a similar manner as the components' locations. To make sure the names stored by the Connection object remain current even if the name of a connected component has just been changed, they are stored as pointers to the actual name variables in the component classes.

Icon and Derived Classes

The icon manipulations all occur by destroying the icon and redrawing it with new specifications.

The engine component icon is a object of class type Engine_Component_Icon. The class Engine_Component_Icon is derived from the Icon class which is essentially the GUI framework's Slab primitive class rewritten for use with a 3D view. Thus, the Engine_Component_Icon has the look of a rectangular push button with a label proclaiming the name of the component. The icon can be moved, activated (lowered), deactivated (raised), selected (highlighted) and deselected (unhighlighted).

To change the name the icon is destroyed, the dimensions required for the new label are calculated and the icon is recreated. The icon is activated, deactivated, selected and unselected by editing the PHIGS graphics structure to make the desired changes.

The icon used for the flow connection type is an object of class type Flow_Connection_Icon, derived from the base class Icon. It is essentially a button much like the one used for engine components. The button is sized to display the positive integer name of the flow station. The way connections to components are indicated is by arrows pointing to and from the connection button. They are also part of the icon. A flow connection icon is shown in Figure 19.

The rotational connection icon, also depicted in Figure 19, consists of two lines: one to each rotational component. The location of the Rotational_Connection_Icon is midway between the two connected engine components. This gives the impression of one continuous, straight line connecting the two components.

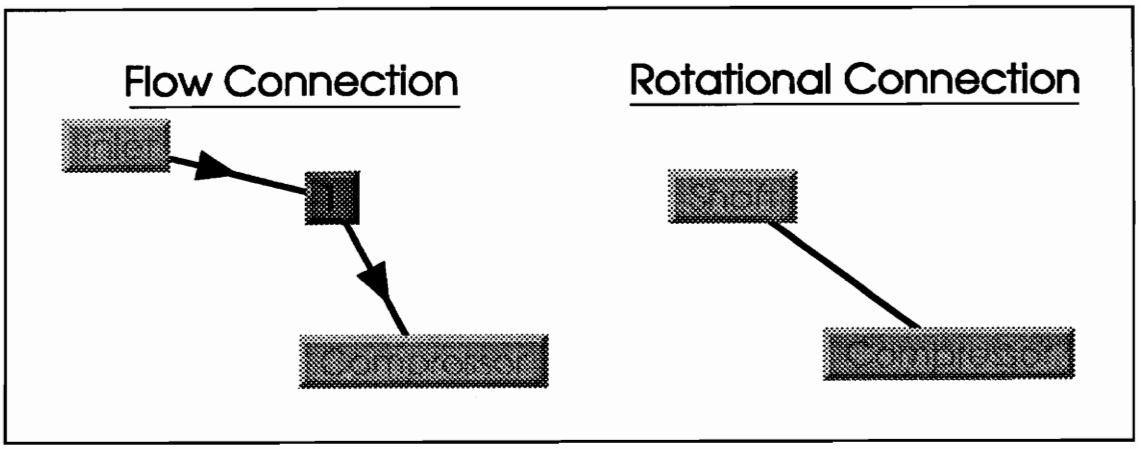


Figure 19 - Connection Icons

Results

A graphical user interface for the aircraft engine design code NEPP was created. Serving as a preprocessor for input file generation and editing it was developed fully in the object-oriented paradigm. While the analysis portion, NEPP, is procedural, it was possible to design and program the graphical user interface in an object-oriented manner.

Figure 20 shows the layout of the work area. The engine configuration shown was created by hand (as opposed to being read from file). No keyboard input was necessary to set up the general configuration. The engine is made up of standard components: Inlet, Compressor, Burner, Turbine, Shaft and Nozzle. A second burner is used as afterburner. Its name will be changed to "Afterburner" as shown in figure 22 to prevent confusion with the first burner. Clearly visible are the numbered flow stations connected to flow components by arrows and the rotational connections represented by lines between compressor and shaft and turbine and shaft. The lower half of the engine was selected as indicated by the highlighting. By picking and dragging any entity all of the highlighted objects moved with it. This is shown by the motion blur caused by lateral dragging of the selected objects.

Figure 21 shows the editing menu which was popped up by clicking in the work area background. Since no menu items are greyed out, conditions are such that every menu item can be picked. From this point

- a new component may be added,

- the selected three components may be deleted,
- a new flow station may be added,
- the selected three flow stations may be deleted,
- objects may be connected or disconnected, and
- all objects may be selected or deselected.

The pop-up menu will disappear on clicking a menu item and the desired action will commence.

Figure 21 also shows the workings of the scroll window. The height of the slider with respect to the vertical scroll bar indicates that only between 60% and 70% of the modeling height is displayed. The slider's position indicates that the current display is showing a section between the top 10% and the lower 30% of the modeling area. Clicking and dragging the scroll bar allows the user to see the top or bottom portion of the display. By resizing the window to full size the user can (in this case) see the whole modeling area. The slider will fill the vertical scroll bar to indicate that the whole height is being displayed. The window can also be kept the same size and the zoom out button (right button in the lower right corner) can be clicked until the whole engine configuration fits into the view.

Figure 22 displays an example of a component data input menu. This menu was invoked by clicking the middle button of the mouse over the inlet icon. The icon responded by lowering itself to indicate the active state. The menu was created using the saved data in its NEPP data array (which happens to be default data in this case). The name field shows the current name of the object, "Inlet". The name can be changed to any string of characters and spaces except to the name of another component. The upstream flow station field shows that the "0" flow station is connected upstream of the inlet. The

downstream flow station field indicates that the "1" flow station is connected downstream. These connections are, of course, also confirmed by the arrows in the schematic. The connections can be deleted by leaving the two input fields blank. The inlet can be connected to different flow stations by substituting numbers of other existing flow stations.

Input to the remaining input fields is monitored by individual error trapping functions for each variable input. An example of an error response is shown in figure 23. In this case the violating variable input will automatically be set to the minimum allowable temperature. The "Ok" button must be clicked again to accept the new field inputs.

Figure 23 shows the options available through the static menu bar at the top of the configuration window. Object creation and storage can be accessed by clicking "Objects". This menu item expands to "Engine Objects..." accessing whole engine configurations and "Engine Component Objects..." accessing individual component storage and creation. In the sample screen an engine configuration was loaded. The message stems from the interface's attempt to recover from a file read error.

Between 20,000 and 25,000 lines of code make up the basic foundation of the GUI. The interface is programmed to facilitate extensions and enhancements. Further error trapping, file storage functionality, off-design input, flight envelope specification and integration with ACSYNT lie in the future.

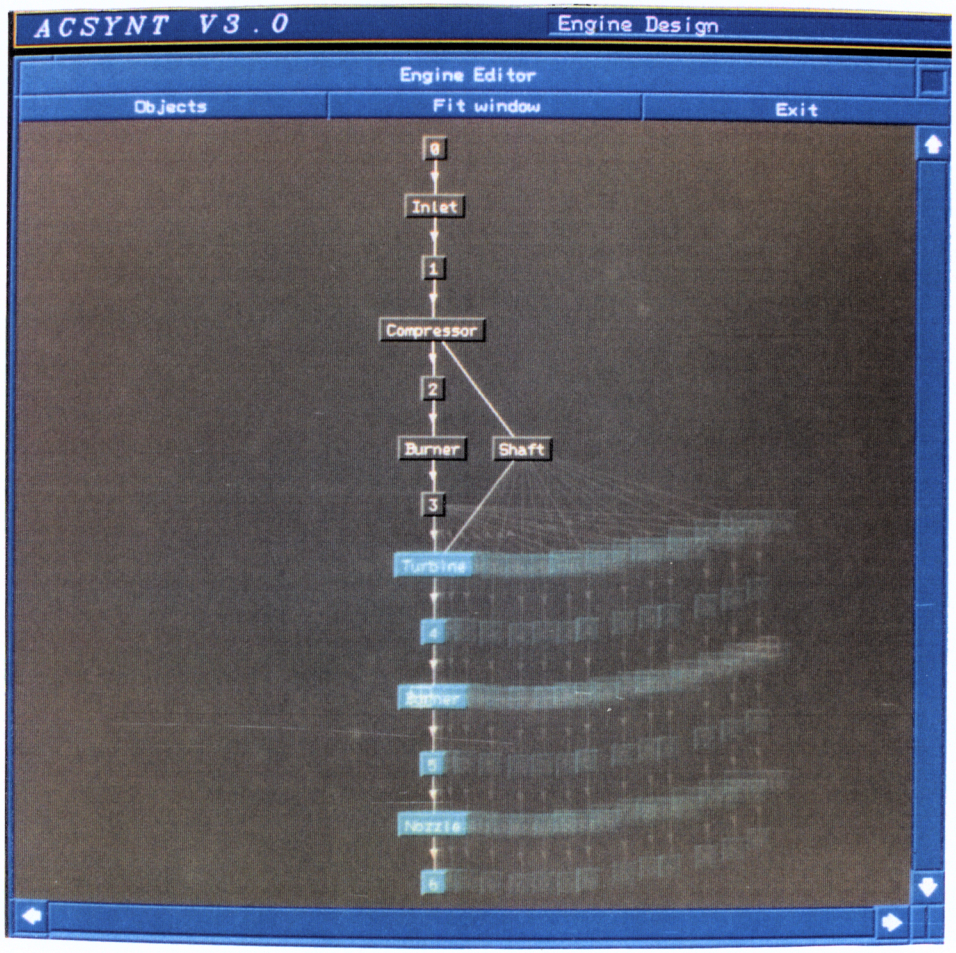


Figure 20 - Photograph 1

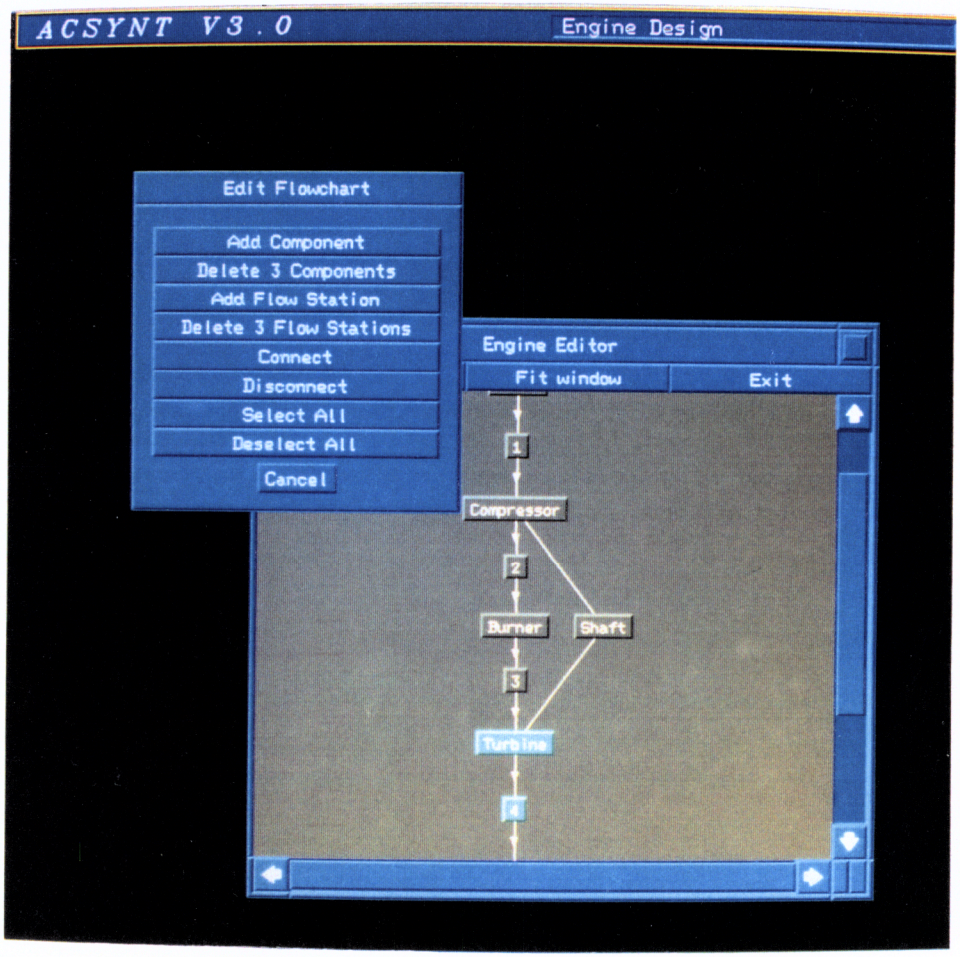


Figure 21 - Photograph 2

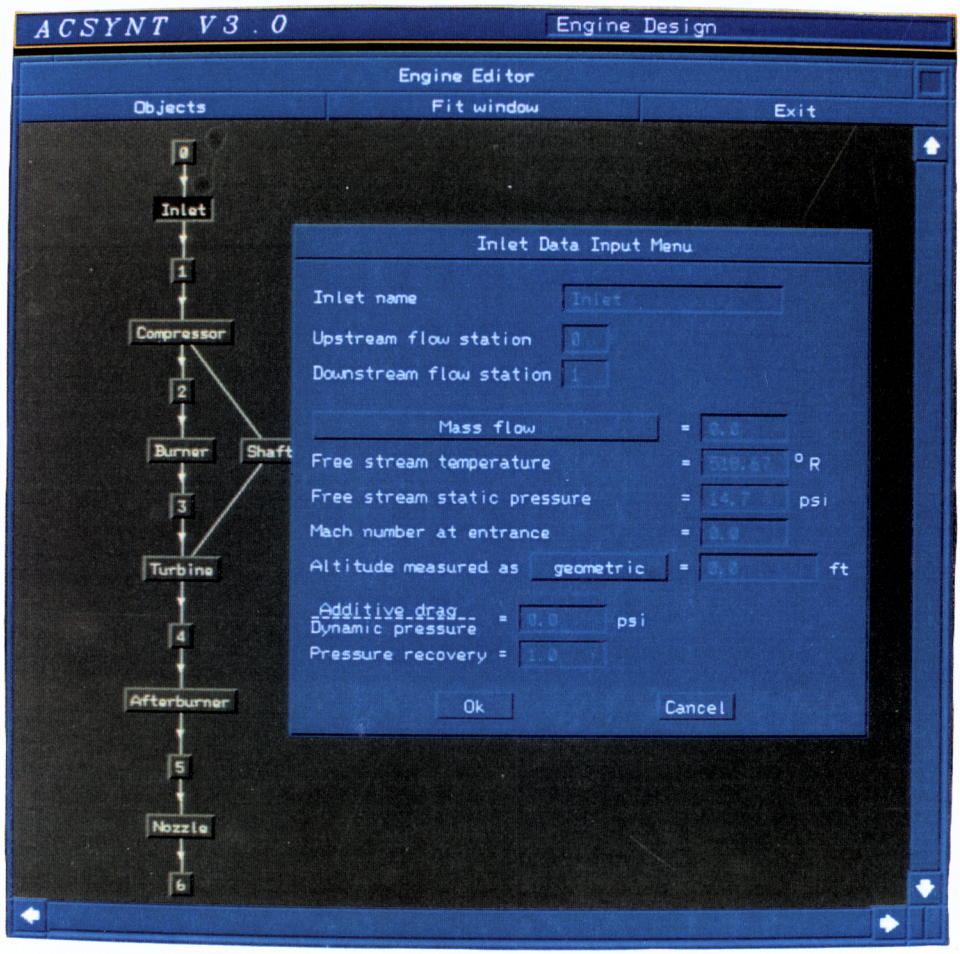


Figure 22 - Photograph 3

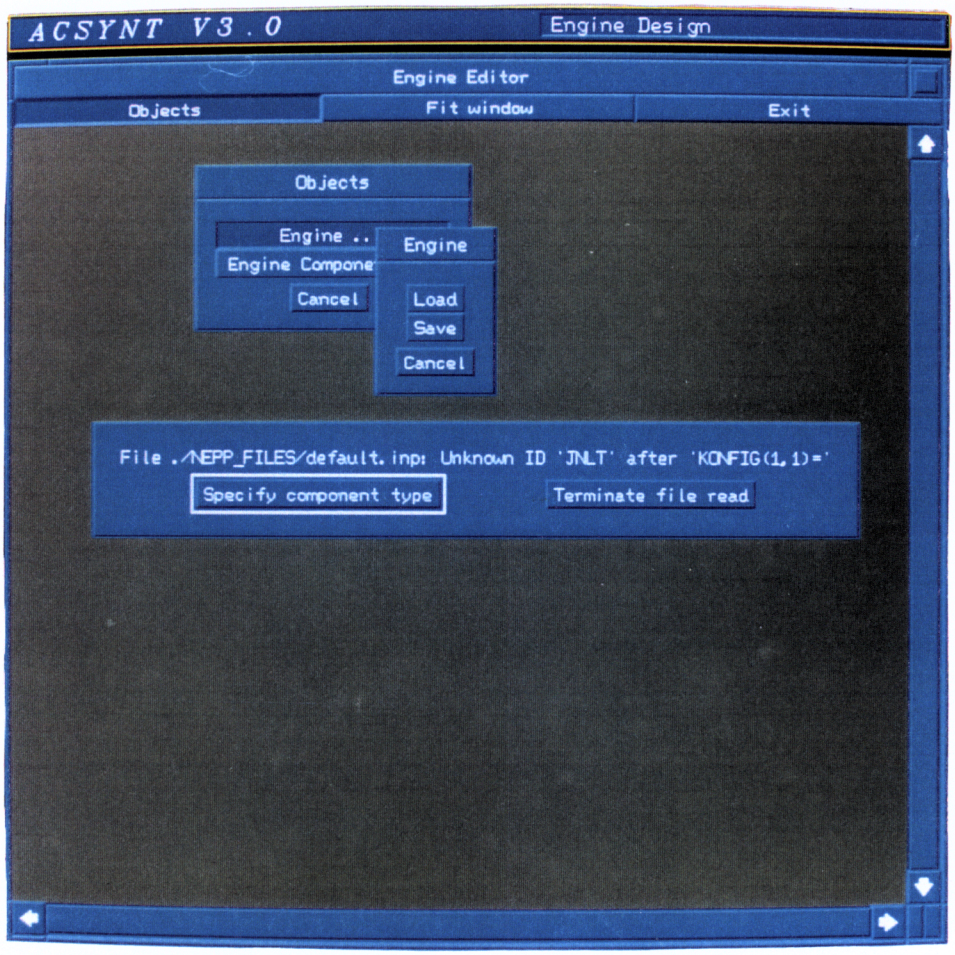


Figure 23 - Photograph 4

Conclusions

The goal of this work was to design and implement a user interface for the aircraft engine cycle analysis program, NEPP, to be used with the interactive CAD aircraft conceptual design program, ACSYNT. The following objectives set forth at the beginning of this thesis have been addressed:

1. Flexibility and Ease of Use
2. Device Independence and Ease of Integration
3. Extensibility and Maintainability

A class hierarchy was created to parallel the physical aircraft engine model. By adhering strictly to this parallel, it was possible to create a group of classes which can be re-used and integrated in the ACSYNT program. The classes can be extended to represent other information modelled in an object-oriented ACSYNT. This might include an engine geometry and location or sizing and weights data or methods. The intuitive graphical user interface to aircraft engine data and methods is contained in the Configuration_Window class. It uses the aircraft engine classes to provide for easy manipulation of NEPP inputs.

By designing complete, well-rounded classes it was possible to keep the external, visible operation of the interface very flexible. Familiar features as found in other GUI's, such as Motif, were used throughout the program. Since only a relatively sparse set of GUI menu items and features was provided with the GUI framework, a lot of supporting work had to

be done to create the necessary utilities needed for this work. These utilities go towards completing the GUI toolkit for future programmers.

Device independence and ease of integration were ensured by using standard or soon-to-be-standard programming languages and graphics languages for every part of the program. Where it was not possible to avoid system dependent programming the code in question was carefully isolated and localized to be easily replaced for use in different hardware. This is the case where the file system is queried or manipulated.

The object-oriented graphical user interface for the NASA Engine Performance Program, even during its development, has exhibited the advantages of the object-oriented software development approach. Reusability, extensibility, and maintainability were demonstrated repeatedly.

Reusability was most obviously displayed during the major restructuring phases in the project. Even though the overall class structure was revised, many of the underlying objects could be reused as they were. This was conceptually akin to writing two separate programs with the same resources of classes and functions. The specialized error trapping methods, for example, were kept intact throughout. Except for changes in those methods which tapped directly into the functionalities of the restructured classes, the main configuration window class remained untouched. Whatever changes were made were insulated within the classes.

Extensibility was also seen throughout the design and coding of the specialized component classes. Since each component (Inlet, Duct, Compressor, etc.) **is** an Engine_Component, almost all methods dealing with components could be generically programmed to refer to

engine components rather than inlets, ducts or compressors. The only exception was in those cases where the type of a component was determined. Those methods could not be fully contained inside the class because once an engine component is created as being of type Engine_Component it cannot be redefined to any other type.

Maintainability refers to a large extent to the ease with which a program can be debugged. This is a function of how well a programmer can find his way around the program and keep a good grasp on all of the program's aspects. With approximately 25,000 lines of commented code the OO-GUI was already well on its way toward the 50,000 line accepted critical break-off point for procedural programs. With many more tens of thousands of lines of OO-GUI framework, the program easily surpassed that mark. Troubleshooting was an everyday activity during development. After a period of acclimatization, C++ was a pleasure to work with and debug. Due to the encapsulation of data and functions into classes, errors, to a large extent, were also encapsulated. Except for errors concerning memory mishandling, debugging was thus quickly confined to a small portion of code.

References

[Aaro84] Aaron, M., "Corporate Identity for Iconic Interface Design: The Graphics Design Perspective", *IEEE Computer Graphics & Applications*, Vol. 4, No. 12, 1984.

[Bert92] Berton, J. J., Plencner, R. M., "An Interfactive Preprocessor for the NASA Engine Performance Program", NASA TM 105786, 1992.

[Boeh86] Boehm, B., "A Spiral Model of Software Development and Enhancement", *Software Engineering Notes*, vol. 11 (4), August 1986, p.22.

[Booc91] Booch, Grady, Object-Oriented Design with Applications, *The Benjamin/Cummings Publishing Company, Inc.* , 1991.

[Cadd75] Caddy, M. J., and Shapiro, S. R., "NEPCOMP - The Navy Engine Performance Computer Program, Version I", NADC - 74045-30, 1975.

[Cunn92] Cunningham, Steve, et al., ed., Computer Graphics Using Object-Oriented Programming, John Wiley & Sons Inc., New York, New York, 1992.

[Elli90] Ellis, M.A., Stroustrup, B., The Annotated C++ Reference Manual, Addison-Wesley Publishing Co., Reading, Massachusetts, 1990.

[Fisc89] Fischer, Gerhard., "Human-Computer Interaction Software, Lessons Learned, Challenges Ahead", *IEEE Software*, January, 1989, pp. 44-52.

[Fish75] Fishbach, L. H., and Caddy, M. J., "NNEP - The Navy NASA Engine Program", NASA TM - X - 71857, 1975.

[Fish81] Fishbach, L. H., "KONFIG and REKONFIG - Two Interactive Preprocessing Programs to the NAVY/NASA Engine Program (NNEP)", NASA TM-82636, 1981.

[Fish88] Fishbach, L. H., and Gordon, S., "NNEPEQ - Chemical Equilibrium Version of the Navy/NASA Engine Program", NASA TM - 100851, 1988.

[Fole84] Foley, J. D., Wallace, V. L., and Chan, P., "The Human Factors of Computer Graphics Interaction Techniques", *IEEE Computer Graphics & Applications*, Vol. 4, No. 11, 1984.

[Gali83] Galitz, W., "Human Engineering in Screen Design", *Journal of System Management*, May 1983, pp. 6-11.

[Halt85] Halter, R., "Man-Machine Interface Design Challenges", *Design News*, pp. 63-70, August 1985.

[HOOP92] "HOOPS: Sixty Commercial Software Vendors Standardize on the HOOPS Graphics Development System", *EDGE: Work-Goup Computing Report*, August 3, 1992, Vol. 3, No. 115, p. 23.

[Jaya90] Jayaram, S., and Myklebust, A., "Towards a Standardized Environment for the Creation of Design and Manufacturing Software", *Proceedings of the International Conference on Engineering Design*, Dubrovnik, Yugoslavia, August 28-31, 1990.

[Jaya91] Jayaram, S., and Myklebust, A., "The Significance of Standards in CAD Education", *Proceedings of University Programs in Computer-Aided Engineering, Design and Manufacturing (UPCAEDM), 9th Annual Conference*, Provo, Utah, May 16-18, 1991, pp.144-147.

[Jaya92] Jayaram, S., Myklebust, A., and Gelhausen, P., "ACSYNT - A Standards-Based Ssytem for Paramentric Computer Aided Conceptual Design of Aircraft", *Proceedings of the 1992 AIAA Aerospace Design Conference*, Irvine, California, February, 1992.

[Jaya93a] Jayaram, S., and Myklebust, A., "Device-Independent Programming Environments for CAD/CAM Software Creation", *Computer Aided Design*, Vol. 25, No. 2, February, 1993.

[Jaya93b] Jayaram, S., and Myklebust, A., "Evaluating PHIGS for CAD Applications - A Case Study", *1st Annual PHIGS User's Group Conference*, March 21-24, 1993, Orlando, Florida.

[Kell93] Kelly, J., "Rule-Based Fuselage and Spine and Cross-section Methods for Computer Aided Design of Aircraft Components", Thesis - Master of

Science in Mechanical Engineering, Virginia Polytechnic Institute and State University, 1993.

[Lee90] Lee, E., "User-Interface Development Tools", *IEEE Software*, Vol. 7, No. 3, May 1990, pp. 31-36.

[Melt85] Meltzer, R. S., "Industry Traversing 'Last Frontier' of User Interfaces", *Computer Graphics Today*, Vol.2, No. 5, p. 9, May 1985.

[Meye92] Meyers, Scott, Effective C++, 50 Specific Ways to Improve Your Programs and Designs, Addison-Wesley Publishing Company, Reading, Massachusetts, 1992.

[Murp93] Murphy, T., "Jumping Through HOOPS", *Computer Language*, Feb. 1993, Vol. 10, No. 2, p. 17.

[Myer89] Myers, B., "User-Interface Tools: Introduction and Survey", *IEEE Software*, January 1989, pp. 15-23.

[Mykl93] Myklebust, A., Gelhausen, P., et al., ACSYNT Installation Manual and User Guide, Version 2.0, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, 1990.

[Onat79] Onat, E., Klees, G. W., "A Method to Estimate Weight and Dimensions of Large and Small Gas Turbine Engines", NASA CR-159481, 1979.

[Penn92] Pennington, S. L., and Myklebust, A., "A Case Approach to the Integration of CAD/CAM System", *presented at the 5th International Conference on Software Engineering and its Applications*, Toulouse, France, Dec. 7-11, 1992.

[Plen91] Plencner, R M., and Snyder, C. A., "The Navy/NASA Engine Program (NNEP89) - A User's Manual", NASA TM - 105186, Lewis Research Center, Cleveland, Ohio, August 1991.

[Powe90] Powell, James E., Designing User Interfaces, Microtrend Books, San Marcos, CA, 1990.

[Rive93] Rivera, F., "An Object-Oriented Method of Mission Profile Input for Aircraft Design", Thesis - Master of Science in Mechanical Engineering, Virginia Polytechnic Institute and State University, 1993.

[Schi92] Schildt, Herbert, Teach Yourself C++, *Osborne McGraw-Hall*, 1992.

[Schr91] Schrock, E. V., "A PHIGS-Based Spreadsheet for Conceptual Design", Thesis - Master of Science in Mechanical Engineering, Virginia Polytechnic Institute and State University, 1991.

[Schr92] Schrock, E. V., Jayaram, S., and Myklebust, A., "A PHIGS-Based Spreadsheet for Conceptual Design", *presented at and published in the Proceedings of the ASEE 5th International Conference on Engineering Computer Graphics and Descriptive Geometry*, Melbourne, Australia, August 17-21, 1992.

[Spen85] Spencer, R. H., "Interactive Software Design: The Human Touch", *Computers in Mechanical Engineering*, September 1985, pp. 45-48.

[Swez83] Swezey, R. W., Davis, E G., "A Case Study of Human Factors Guidelines in Computer Graphics", *IEEE Computer Graphics & Applications*, November 1983, pp. 21-30.

[Uhor93a] Uhorchak, R. S., "An Object-Oriented Class Library for the Creation of Engineering Graphs", Thesis - Master of Science in Mechanical Engineering, Virginia Polytechnic Institute and State University, 1993.

[Uhor93b] Uhorchack, R. S., Jayaram, S., "An Object-Oriented Class Library for the Creation of Engineering Graphs Using PHIGS", *1st Annual PHIGS User's Group Conference*, March 21-24, 1993, Orlando, Florida.

[Wamp88a] Wampler, S., Myklebust, A., Jayaram, S., and Gelhausen, P., "Improving Aircraft Conceptual Design - A PHIGS Interactive Graphics Interface For ACSYNT", AIAA Paper 88-4481, 1988.

[Wamp88b] Wampler, S., "Development of a CAD System for Automated Conceptual Design of Supersonic Aircaft", Thesis - Master of Science in Mechanical Engineering, Virginia Polytechnic Institute and State University, 1988.

[Wamp91] Wampler, S., "Development of a graPHIGS Based Object-Oriented Graphics System", *Proceedings of the 2nd International graPHIGS User's Group Conference and Workshop*, October, 1991, pp. 145-155.

[Wils91] Wilson, M., and Conway, A., "Enhanced Interaction Styles for User Interfaces", *IEEE Computer Graphics & Applications*, March 1991, pp. 79-90.

[Wiss90] Wisskirchen, Peter, Object-Oriented Graphics - From GKS and PHIGS to Object-Oriented Systems, Springer Verlag, Berlin, 1990.

[Woya92] Woyak, Scott, "A Motif-Like Object-Oriented Interface Framework using PHIGS", Thesis - Master of Science in Mechanical Engineering, Virginia Polytechnic Institute and State University, 1992.

[Woya93] Woyak, S., and Myklebust, A., "A Motif-Like Object-Oriented Interface Framework Using PHIGS", *1st Annual PHIGS User's Group Conference*, March 21-24, 1993, Orlando, Florida.

Appendix A - Detailed C++ Class Descriptions

Class Name: Scroll_Window

Description: The scroll window is a sizable and shiftable menu used to display two-dimensional and three-dimensional geometries. Its features are a horizontal and vertical scroll bar and zoom in/zoom out buttons.

Header File Name: scroll_window.h

Derived from: Geometry_Manager

Member Classes:

Scroll_Bar* vertical_scroll ..... *Menu item used to translate geometry up or down*
Scroll_Bar* horizontal_scroll ..... *Menu item used to translate geometry left or right*
Push_Button* zoom_in ..... *Menu item used to enlarge the geometry*
Push_Button* zoom_out ..... *Menu item used to reduce the geometry*

Member Data:

private:

float old_geo_width ..... *Previous saved width of geometry view*
float old_geo_height ..... *Previous saved height of geometry view*
float geo_width ..... *Current width of geometry view*
float geo_height ..... *Current height of geometry view*
float old_horizontal_movement ..... *Previous saved left-right movement of modeling area with respect to display area*
float old_vertical_movement ..... *Previous saved up/down movement of modeling area with respect to display area*
float old_x_center ..... *Previous saved x location of the display center on the modeling area*
float old_y_center ..... *Previous saved y location of the display center on the modeling area*

float x_center_offset.....*x distance from the reference point of the display area to the reference point of the modeling area*

float y_center_offset.....*y distance from the reference point of the display area to the reference point of the modeling area*

float scale.....*conversion factor from display dimensions to modeling dimensions*

double stored_magnification.....*The user controlled magnification is stored in this variable as the log_{10} so that accuracy of the magnification is not lost for magnifications less than 1.0*

protected:

float model_x_min.....*Minimum x coordinate of the displayed geometry in modeling coordinates.*

float model_y_min.....*Minimum y coordinate of the displayed geometry in modeling coordinates.*

float model_width.....*Maximum width of the displayed geometry in modeling coordinates.*

float model_height.....*Maximum height of the displayed geometry in modeling coordinates.*

float display_width.....*Width of the display area in modeling coordinates*

float display_height.....*Height of the display area in modeling coordinates*

float horizontal_value.....*Percent value of the horizontal scroll bar*

float vertical_value.....*Percent value of the vertical scroll bar*

double magnification.....*Current user controlled magnification*

double zoom_factor.....*Factor to magnify the display area by each time a zoom is requested*

float horizontal_scroll_increment.....*The increment in modeling by coordinates which to scroll the display when the left or right button of the scroll bar is pressed*

float vertical_scroll_increment.....*The increment in modeling coordinates by which to scroll the display when the up or down button of the scroll bar is pressed*

float horizontal_scroll_percent_increment.....*The horizontal scroll increment as a percent value of the modeling width*

float vertical_scroll_percent_increment.....*The vertical scroll increment as a percent value of the modeling height*

float horizontal_page_scroll_percent_increment.....*The horizontal scroll increment as a percent value of the modeling width. This increment is used when either side of the slider slot is clicked. Usually this increment should be set to scroll the display almost one full screen at a time.*

float vertical_page_scroll_percent_increment.....*The vertical scroll increment as a percent value of the modeling height. This increment is used when either side of the slider slot is clicked. Usually this increment should be set to scroll the display almost one full screen at a time.*

int ETC_requested.....*Flag specifying whether immediate mode graphics were requested by the user*

Member Functions:

virtual:

void create_additional_components(void)

Allows the user of the class to add more menu items to the scroll window

void delete_additional_components(void)

Used by the destructor to delete the menu items created by create_additional_components

void process_additional_event(Event *_event)

Processes the events generated by the additional menu items

_event ..... *(Returned) The event information object maintained by the interface*

void set_additional_differences(void)

Makes room for the new additional components in the space between the geometry view and the border of the scroll window

void process_geometry_view(int _choice, Ppoint3 *_locator_pos, int _view_index, Ppick_path *_pick_path, Event *_event)

This is the event handler for events generated inside the geometry view.

_choice ..... *(Returned) Type of event*

_locator_pos ..... *(Returned) A PHIGS structure for the locator position*

_view_index ..... *(Returned) The PHIGS view index*

_pick_path ..... *(Returned) The PHIGS structure containing the pick information*

_event ..... *(Returned) The event information object maintained by the interface*

void turn_ETC_on(void)

This function turns immediate mode graphics on.

void turn_ETC_off(void)

This function turns immediate mode graphics off.

private:

void initializer(float _x, float _y, int _horizontal_alignment, int _vertical_alignment, float _width, float _height, float _display_dimension, int _specified_dimension, float _model_width, float _model_height)

Takes care of initializations common to all constructors

_x ..... *x-position of window*

_y ..... *y-position of window*

_horizontal_alignment ..... *horizontal reference position on window (LEFT, CENTER, or RIGHT)*

_vertical_alignment ..... *vertical reference position on window (TOP, CENTER, or BOTTOM)*

_width ..... *width of window in normalized projection coordinates (0 to 1)*

_height ..... *height of window in normalized projection coordinates (0 to 1)*

_display_dimension ..... *height or width of display view in modeling coordinates*

_specified_dimension ..... *refers to previous argument: was HEIGHT or WIDTH specified*

_model_width ..... *width of the model to be displayed*

_model_height ..... *height of the model to be displayed*

protected:

void synchronize_scroll_window(int _perform_flag)

Synchronizes the position of the display with the position of the scroll bars

_perform_flag ..... *Update the screen? (PERFORM or DO_NOT_PERFORM)*

void synchronize_display_with_horizontal_scroll_bar(int _perform_flag)

Synchronizes the position of the screen with respect to the position of the horizontal scroll bar

int _perform_flag ..... *Update the screen? (PERFORM or DO_NOT_PERFORM)*

void synchronize_display_with vertical_scroll_bar(int _perform_flag)

Synchronizes the position of the screen with respect to the position of the horizontal scroll bar

int _perform_flag ..... *Update the screen? (PERFORM or DO_NOT_PERFORM)*

void synchronize_horizontal_scroll_bar_with_display(int _perform_flag)

Synchronize the position of the horizontal scroll bar with respect to the position of the display

int _perform_flag ..... *Update the screen? (PERFORM or DO_NOT_PERFORM)*

void synchronize_vertical_scroll_bar_with_display(int _perform_flag)

Synchronize the position of the vertical scroll bar with respect to the position of the display

int _perform_flag ..... *Update the screen? (PERFORM or DO_NOT_PERFORM)*

void calculate_page_scroll_percent_increments(void)

Calculate the horizontal and vertical scroll percent increments corresponding to one display area.

public:

Scroll_Window(SCROLL_WINDOW_DECLARATIONS)
Scroll_Window(SCROLL_WINDOW_DECLARATIONS, int dummy)
Scroll_Window(SCROLL_WINDOW_DECLARATIONS, char *_title)
Scroll_Window(SCROLL_WINDOW_DECLARATIONS, char *_title, float _character_height)
Scroll_Window(SCROLL_WINDOW_DECLARATIONS, float _thickness, char *_title)
Scroll_Window(SCROLL_WINDOW_DECLARATIONS, float _thickness, char *_title, float _character_height)
Scroll_Window(SCROLL_WINDOW_DECLARATIONS, float _thickness, char *_title, float _character_height, Color_Group *_color)

These are the available constructors. SCROLL_WINDOW_DECLARATIONS is defined to be the list of arguments needed by the initializer. If the second form is used, this class acts as a plain Geometry_Manager class.

_title ..... *Title of window*
_character_height ..... *Height of title characters*
_thickness ..... *Thickness of bevel shadows*
_color ..... *An object containing color information to be used for the scroll window*

void set_2D_3D_differences_rel(float _x_difference, float _y_difference, float _width_difference, float _height_difference, int _specified_dimension, int _perform_flag)

This function allows the placement and sizing of the geometry view with respect to the current geometry view.

_x_difference ..... *Distance from the previous x-location to the new x-location of the geometry view*
_y_difference ..... *Distance from the previous y-location to the new y-location of the geometry view*
_width_difference ..... *Difference in width between the previous and desired geometry view*
_height_difference ..... *Difference in height between the previous and desired geometry view*

_specified_dimension.....*HORIZONTAL or VERTICAL - the dimension that should be scaled to, to set the size of the display area.*

_perform_flag.....*Update display? PERFORM or DO_NOT_PERFORM*

void set_2D_3D_differences_abs(float _x_difference, float _y_difference, float _width_difference, float _height_difference, int _specified_dimension, int _perform_flag)

This function allows the placement and sizing of the geometry view with respect to the borders of the scroll window.

_x_difference.....*Distance from the inside of the left border to the new x-location of the geometry view*

_y_difference.....*Distance from the bottom of the titlebar to the new y-location of the geometry view*

_width_difference.....*Difference in width between the width of the scroll window from border to border and the desired width of the geometry view*

_height_difference.....*Difference in height between the height of the scroll window from bottom of titlebar to top of lower border and the desired height of the geometry view*

_specified_dimension.....*HORIZONTAL or VERTICAL - the dimension that should be scaled to, to set the size of the display area.*

_perform_flag.....*Update display? PERFORM or DO_NOT_PERFORM*

float get_horizontal_scroll_percent_value(void)

Return the current percentage value of the horizontal scroll bar.

float get_vertical_scroll_percent_value(void)

Return the current percentage value of the vertical scroll bar.

float get_horizontal_scroll_increment(void)

Return the current scroll increment of the horizontal scroll bar.

float get_vertical_scroll_increment(void)

Return the current scroll increment of the vertical scroll bar.

float get_horizontal_scroll_percent_increment(void)

Return the current scroll increment in percent of the horizontal scroll bar.

float get_vertical_scroll_percent_increment(void)

Return the current scroll increment in percent of the vertical scroll bar.

float get_model_width(void)

Return the current model width.

float get_model_height(void)

Return the current model height.

float get_zoom_factor(void)

Return the current zoom factor.

float get_magnification(void)

Return the current magnification.

void set_horizontal_scroll_percent_value(float _percent, int _perform_flag)

Manually set the horizontal scroll bar percentage.

_percent.....*New percent value*

_perform_flag.....*Update display? PERFORM or DO_NOT_PERFORM*

void set_vertical_scroll_percent_value(float _percent, int _perform_flag)

Manually set the vertical scroll bar percentage.

_percent.....*New percent value*
_perform_flag.....*Update display? PERFORM or DO_NOT_PERFORM*

void set_horizontal_scroll_increment(float _horizontal_scroll_increment)
Set the horizontal scroll bar increment.
_horizontal_scroll_increment.....*New scroll increment in modeling coordinates*

void set_vertical_scroll_increment(float _vertical_scroll_increment)
Set the vertical scroll bar increment.
_vertical_scroll_increment.....*New scroll increment in modeling coordinates*

void set_horizontal_scroll_percent_increment(float _horizontal_scroll_percent_increment)
Set the horizontal scroll bar increment as a percent value.
_horizontal_scroll_percent_increment.....*New scroll increment as a percent value*

void set_vertical_scroll_percent_increment(float _vertical_scroll_percent_increment)
Set the vertical scroll bar increment as a percent value.
_vertical_scroll_increment.....*New scroll increment as a percent value*

void set_model_x_min(float _model_x_min, int _perform_flag)
Set the minimum x-coordinate of the modeling space.
_model_x_min.....*New minimum x in modeling coordinates*
_perform_flag.....*Update display? PERFORM or DO_NOT_PERFORM*

void set_model_y_min(float _model_y_min, int _perform_flag)
Set the minimum y-coordinate of the modeling space.
_model_y_min.....*New minimum x in modeling coordinates*
_perform_flag.....*Update display? PERFORM or DO_NOT_PERFORM*

void set_model_width(float _model_width, int _perform_flag)
Set the width of the modeling space.
_model_width.....*New model width in modeling coordinates*
_perform_flag.....*Update display? PERFORM or DO_NOT_PERFORM*

void set_model_height(float _model_height, int _perform_flag)
Set the height of the modeling space.
_model_height.....*New model height in modeling coordinates*
_perform_flag.....*Update display? PERFORM or DO_NOT_PERFORM*

void set_zoom_factor(float _zoom_factor)
void set_magnification(float _magnification, int _perform_flag)

Class Name: Scroll_Bar

Description: The scroll bar can have either vertical or horizontal orientation. It consists of a slider with labeled push buttons at either end. The scroll bar is used to change a value continuously using the slider or incrementally using the push buttons.

Header File Name: scroll_bar.h

Derived from: Menu_Item

Member Classes:

Color_Group* color	*Color information object for scroll bar*
Event* change_event	*The event object generated by the scroll bar*
Slider* slider	*Vertical or horizontal slider part of the scroll bar, used to change the value continuously*
Arrow* high_arrow	*Label for high_button*
Push_Button* high_button	*Button on top or right of the slider used to increase the value incrementally*
Arrow* low_arrow	*Label for low_button*
Push_Button* low_button	*Button on bottom or left of the slider used to decrease the value incrementally*

Member Data:

private:

int erase_color_index	*This is the background color index used by immediate mode graphics*
float shadow_thickness	*Thickness of the bevel shadow*
float percent_value	*Percent value of the scroll bar*
float event_data[2]	*The value and delta since the last movement*
float saved_value	*The previous saved value*
float scroll_increment	*Step by which the value is incremented or decremented each time a button is clicked*
float page_scroll_increment	*Step size by which the value is incremented or decremented each time the slider slot on either side of the slider slab is picked. This is used for scrolling page by page.*
float x_min, x_max, y_min, y_max	*Boundary coordinates used for all of the components of the scroll bar*

float total_x, total_y ..... *Coordinates of reference point of the entire scroll bar*

float total_width, total_height ..... *Dimensions of the entire scroll bar*

float length, thickness ..... *Dimensions of the scroll bar - length corresponds to total_width for a horizontal scroll bar, etc.*

int horizontal_alignment, vertical_alignment ..... *Position of the reference point of the scroll bar: horizontal_alignment can have the values LEFT, CENTER, RIGHT; vertical_alignment can have the values TOP, CENTER, BOTTOM*

int orientation ..... *Orientation of the scroll bar: VERTICAL or HORIZONTAL*

int event_time ..... *Dictates whether the processing loop should respond immediately to any change in the scroll bar or whether it should wait to update values until the **ok_close** function is called to terminate the menu process.*

int event_id ..... *Scroll bar event id*

float slider_slot_x, slider_slot_y ..... *Location of the upper left corner of the slider slot*

float high_button_x, high_button_y ..... *Location of the upper left corner of the high end push button*

float low_button_x, low_button_y ..... *Location of the upper left corner of the low end push button*

Member Functions

virtual:

void manage(PHIGS_View* _view, float _priority)

Tell the menu manager to manage slider and push buttons.

_view ..... *PHIGS view information of the view in which the scroll bar resides*

_priority ..... *PHIGS structure priority*

void unmanage(void)

Tell the menu manager not to manage the slider and push buttons. This effectively releases the scroll bar from any event loop.

int is_cursor_over_menu_item(Ppoint3* _locator_pos)

Detect whether the mouse cursor is over the menu item. This function returns TRUE if the location of the cursor is within the bounds of the menu item.

_locator_pos ..... *Pointer to PHIGS structure containing the x, y and z coordinates of the current cursor (or locator) position.*

void process_from_mouse(int _choice, Ppoint3* _locator_pos, int _view_index, Ppick_path* _pick_path, Event* _event)

This is the event handler for mouse events. It handles clicks and drags over the scroll bar. The events which are recognized by the function include all of the events recognized by the push buttons and the slider. If a push button was clicked the value is incremented or decremented. If the slider slot was clicked the value is incremented or decremented by page_scroll_increment. If the slider was dragged, the value is changed by an amount proportional to the change in the slider position.

_choice ..... *Type of event*

_locator_pos ..... *Pointer to PHIGS structure containing the x, y and z coordinates of the current cursor position.*

_view_index ..... *PHIGS view number*
_pick_path ..... *Pointer to PHIGS structure containing the path information of the pick*
_event ..... *Pointer to an event information object.*

void process_from_keyboard(int _stroke, Event* _event)

This is the event handler for keyboard events. Since the scroll bar cannot be highlighted keyboard events can not be implicitly directed towards it. This function may have to be called explicitly by the application's event loop. The function recognizes up, down, left and right arrow keys.

_stroke ..... *Id of the pressed key*
_event ..... *Pointer to an event information object*

int is_highlightable(void)

The scroll bar is not highlightable.

void highlight(int _perform_flag)

This function remains a dummy function.

_perform_flag ..... *Update display? PERFORM or DO_NOT_PERFORM*

void unhighlight(int _perform_flag)

This function remains a dummy function

_perform_flag ..... *Update display? PERFORM or DO_NOT_PERFORM*

int get_highlight_condition(void)

Since the scroll bar is not highlightable this function returns NO

void save_state(void)

This function saves the value of the scroll bar as saved_value.

void revert_to_saved_state(void)

Set the scroll bar back to the saved_value.

void generate_delayed_events(Event* _event)

This function generates events if the scroll bar was managed in delayed event mode. This function gets called by the **ok_close** function.

_event ..... *Pointer to an event information object*

private:

***SCROLL_BAR_ARGS** is defined as the following list of arguments:*

int _event_id, float _x, float _y, int _horizontal_alignment, int _vertical_alignment, float _length, float _thickness, int _orientation

_event_id ..... *Event id of the scroll bar*
_x ..... *x-location of the scroll bar reference point*
_y ..... *y-location of the scroll bar reference point*
_horizontal_alignment ..... *Position of the reference point on the scroll bar: LEFT, CENTER, RIGHT*
_vertical_alignment ..... *Position of the reference point on the scroll bar: TOP, CENTER, BOTTOM*
_length ..... *Length of the scroll bar: This is the width for a horizontal scroll bar and the height for a vertical scroll bar*
_thickness ..... *Thickness of the scroll bar: This is the height for a horizontal scroll bar and the width for a vertical scroll bar*
_orientation ..... *Orientation of the scroll bar: VERTICAL or HORIZONTAL*

void initializer(SCROLL_BAR_ARGS, float _shadow_thickness)

This function takes care of initializations and procedures which need to be carried out by all constructors.
_shadow_thickness ..... *Thickness of the bevel shadow*

void create_components(void)

This function is called by the constructor to create the slider, push buttons and arrows

void set_locations(void)

Calculate the locations of all components.

void set_min_and_max(void)

Calculate the minima and maxima of all components.

void h_left(void), h_center(void), h_right(void)

Calculate the x-coordinates of components at left, center and right of the component.

void v_top(void), v_center(void), v_bottom(void)

Calculate the y coordinates of components at top, center and bottom of the component.

void display_change(void)

Display any change in the slider location.

public:

The following constructor functions are available:

Scroll_Bar(SCROLL_BAR_ARGS)

Scroll_Bar(SCROLL_BAR_ARGS, Color_Group* _color)

Scroll_Bar(SCROLL_BAR_ARGS, float _shadow_thickness)

Scroll_Bar(SCROLL_BAR_ARGS, float _shadow_thickness, Color_Group* _color)

Scroll_Bar(SCROLL_BAR_ARGS, float _shadow_thickness, float _scroll_increment, float _page_scroll_increment)

Scroll_Bar(SCROLL_BAR_ARGS, float _shadow_thickness, float _scroll_increment, float _page_scroll_increment, Color_Group* _color)

_color ..... *Color information object for scroll bar*

_shadow_thickness ..... *Thickness of bevel shadow*

_scroll_increment ..... *Step size of scroll bar's push button response*

_page_scroll_increment ..... *Step size of scroll bar's slider slot pick response*

float get_percent_value(void)

The scroll bar maintains a value from 0 to 100 depending on the current position of the slider within the menu item. This function returns this value.

void set_percent_value(float _percent_value, int _perform_flag)

The value and, with it, the position of the slider can be set from outside the Scroll_Bar class using this function.

_percent_value ..... *The new value and relative position of the slider*

_perform_flag ..... *Update display? PERFORM or DO_NOT_PERFORM*

void set_scroll_increment(float _scroll_increment)

Set the increment by which the percent value is to be decremented or incremented when one of the push buttons is clicked.

_scroll_increment ..... *The new scroll increment value*

void set_page_scroll_increment(float _page_scroll_increment)

Set the increment by which the percent value is to be decremented or incremented when the slider slot is clicked on either side of the slider.

_page_scroll_increment ..... *The new page scroll increment value*

void set_change_event_id(int _id)

The event id for the scroll bar can be set using this function.

_id ..... *The new event id*

void set_event_time_to_immediate(void)

Specify that events are to be responded to immediately.

void set_event_time_to_delayed(void)

Specify that the value is to be updated when the menu manager's **ok_close** function closes the menu.

void turn_ETC_on(void)

This function turns on immediate graphics mode if available.

void turn_ETC_off(void)

Turn off immediate graphics mode.

void set_color(Color_Group* _color)

The color of the scroll bar can be set after the constructor has been called using this function.

_color ..... *Pointer to color information object for scroll bar*

void set_slider_length_percent(float _percent, int _perform_flag)

The length of the slider with respect to the length of the slot can be set using this function. For displays this can be used to let the slider represent the displayed portion of the total available information.

_percent ..... *New length of slider in percentage of the slot length*

_perform_flag ..... *Update display? PERFORM or DO_NOT_PERFORM*

Class Name: Configuration_Window

Description: A configuration window is a Scroll_Window object with the ability to display and edit an NEPP aircraft engine configuration its 3D display view. The configuration is mad up of pickable components, flow stations and their connecting lines and arrows. The components and flow stations are labeled push buttons.

- Clicking on a component or flow station selects it.
- Clicking the middle button on a component or flow station activates the input menu associated with it.
- Clicking and dragging a component or flow station moves it.
- Clicking in the blank background activates a pop-up menu for editing the configuration.
- Further functions are available in a menu bar across the top of the window.

Header File Name: configuration.h

Derived from: Scroll_Window

Member Classes:

private:

Push_Button* object_button ..... *Menu button at top of window - It offers storage options for engine and engine component objects.*

Push_Button* fitted_window_button ..... *Menu button at top of window - It is a toggle between automatically fitting the geometry to the position of the scroll bars and leaving it fixed.*

Push_Button* exit_button ..... *Menu button at top of window - Clicking this button exits the configuration window after a confirmation message has been confirmed.*

public:

Engine_Component* components ..... *Linked list of engine components maintained by the configuration window*

Connection* connections ..... *Linked list of engine component connections maintained by the configuration window*

Graphics_Info* graphics_info ..... *This is a graphics information object containing relevant information about the window. It can be used to send graphics information to the other graphical elements in bulk.*

Member Data:

private:

int NEPP_exit_flag ..... *This is a flag set for the external processing loop to determine when the configuration should be exited and deleted. This must be done because the window cannot kill itself.*

float x_min, x_max, y_min, y_max ..... *These are the minima and maxima of the displayed geometry in modeling coordinates. They are used for autocentering and fitting.*

float pop_x, pop_y ..... *These variables are used for popping items on the screen. They are set to the location where the last cursor activity occurred.*

char window_mode* ..... *This string is used as the label for the fitted_window_button. It can be either "Fit Window" or "Fix Window"*

Member Functions:

CONFIGURATION_WINDOW_DECLARATIONS is defined as the following list of arguments:

float _x, float _y, int _horizontal_alignment, int _vertical_alignment, float _width, float _height, int _display_dimension, int _specified_dimension, float _model_width, float _model_height

_x ..... *x-position of window*

_y ..... *y-position of window*

_horizontal_alignment ..... *horizontal reference position on window (LEFT, CENTER, or RIGHT)*

_vertical_alignment ..... *vertical reference position on window (TOP, CENTER, or BOTTOM)*

_width ..... *width of window in normalized projection coordinates (0 to 1)*

_height ..... *height of window in normalized projection coordinates (0 to 1)*

_display_dimension ..... *height or width of display view in modeling coordinates*

_specified_dimension ..... *refers to previous argument: was HEIGHT or WIDTH specified*

_model_width ..... *width of the model to be displayed*

_model_height ..... *height of the model to be displayed*

void initializer(CONFIGURATION_WINDOW_DECLARATIONS)

This function is called by all Configuration_Window constructors. It contains initializations common to all of them. It initializes the object variables by setting them to NULL, sets the "pop coordinates" to zero and sets the background color of the display.

void set_additional_differences(void)

Room has to be made to fit the menu bar at the top of the window between the title bar and the display view. This function calculates the space needed and reduces the size of the display view accordingly. This function is called by the base class when the window is created.

void create_additional_components(void)

The additional menu items at the top of the window are created in this function. This function is called by the base class when the window is created.

void delete_additional_components(void)

The additional menu items are deleted in this function when the destructor is called.

void process_additional_event(Event* _event)

Any events generated by the additional components are processed in this event handler. This function is called from the base class event handler.

_event ..... *Object with all current event information*

void pop_up_edit_menu(float _x, float _y)

A pop-up menu can be activated to access editing options. The function first checks the current state of the configuration to decide which menu items should be deactivated and what the label for each menu item should say. For example if no components are highlighted the "delete" menu item is deactivated. If six components are highlighted then the menu item's label reads, "Delete 6 components". The function then creates the menu, manages it and enters an event loop. The loop is exited and the pop-up is deleted when a button is clicked.

_x, _y ..... *x and y coordinates of new location for the pop-up menu*

void wait_for_pick(Ppick_path* _pick_path)

This function loops until a pick occurs. All other events are flushed from the event queue. The pick information is returned in the argument _pick_path.

_pick_path ..... *Object with information about the pick*

void update_geometry(void)

If the display mode is "Fit Geometry" then the display is updated and the function is exited. Otherwise find the absolute minima and maxima coordinates of the model. Then relay that information to the base class to induce it to update the size of the scroll bars and the position of the geometry. Lastly update the display.

protected:

void process_geometry_view(int _choice, Ppoint3* _locator_pos, int _view_index, Ppick_path* _pick_path, Event* _event)

This is the main processing loop for the geometry display. In pseudo code it looks as follows:

If nothing was picked (i.e. the mouse button was pressed over the blank background) then
    pop up the editing menu
Else if something was picked
    get the component or connection corresponding to the structure id
    If the left button was pressed then
        While the mouse button is not released
            Sample the location
            If the mouse was dragged then
                move the component
            If object was only clicked, not dragged then
                select-toggle the component (this highlights the component)
    Else if the middle button was pressed then
        While the mouse button is not released

Sample the location
If the mouse is located over the icon then
activate the icon
Else
deactivate the icon
If the icon is active then
Pop-up the input menu for the component or connection
deactivate the icon

_choice ..... *The type of event (ex.: LEFT_DOWN or MIDDLE_DOWN)*

_locator_pos ..... *PHIGS structure with x, y and z coordinates of the current locator position*

_view_index ..... *PHIGS view number in which pick event occurred*

_pick_path ..... *PHIGS data structure with information about the pick*

_event ..... *Object with current event information*

void add_component(Component_Type _type)

This function adds an engine component to the linked list of displayed components. It is outlined as follows:

Create a generic engine component
Set the component type by using the object's set_type function
Create an engine_component according to this new type:
Currently the engine_component is the generic base class. To change to the specific, derived class, (for example, inlet or duct) the generic component must be deleted and the derived component must be created.
Add the component to the linked list of engine components
Resize window and update the geometry

_type ..... *Engine component type - This can be one of the following:*
- *NO_TYPE*
- *INLET_TYPE*
- *DUCT_TYPE*
- *BURNER_TYPE*
- *GAS_GENERATOR_TYPE*
- *WATER_INJECTOR_TYPE*
- *FAN_TYPE*
- *COMPRESSOR_TYPE*
- *TURBINE_TYPE*
- *HEAT_EXCHANGER_TYPE*
- *FLOW_SPLITTER_TYPE*
- *FLOW_MIXER_TYPE*
- *EJECTOR_TYPE*
- *NOZZLE_TYPE*
- *LOAD_TYPE*
- *PROPELLER_TYPE*
- *SHAFT_TYPE*

void remove_components(int _confirm_flag)

Engine components can be removed using this function. When this function gets called all selected (highlighted) components are removed. The confirmation flag, _confirm_flag, indicates whether or not error checking and a user warning should take place. This is necessary for cases where a component is removed behind the scenes.

_confirm_flag ..... *Confirmation flag - CONFIRM or DO_NOT_CONFIRM*

int add_flow_connection(void)

This function adds a flow station to the linked list of connections. It assigns a name by going through the list to find the first unused positive integer.

int add_rotational_connection(void)

This function adds a rotational connection to the linked list of connections. It assigns a name by going though the list to find the first unused, negative, non-zero integer.

void remove_connections(int _confirm_flag)

Connections can be removed using this function. When this function gets called all selected (highlighted) connections are removed. The confirmation flag, _confirm_flag, indicates whether or not error checking and a user warning should take place. This is necessary for cases where a connection is removed behind the scenes.

_confirm_flag ..... *Confirmation flag - CONFIRM or DO_NOT_CONFIRM*

void connect_components(void)

This function connects two components or a component and a flow station. It does preliminary error checking to make certain that the two items are compatible. The following pseudo-code explains the procedure:

```
If there are not enough items to make a connection then
    display an error message and exit the function.
Loop while the user picks the two items to connect
    If the pick was in the background, exit the function
    else a component or a connection was picked...
        If the pick was a connection then
            set a flag to remember that it can only be connected upstream of a component
    If the second pick was in the background, exit the function
    else a component or connection was picked...
        If this item is the same component as the first, issue error message and exit.
        If the first item and this item are connections, issue error message and exit because
            two connections cannot be connected.
Connect the objects...
Figure out whether a rotational or flow connection is being done.  Then do preliminary
 error trapping to make sure each object allows a connection to the other.
To do a ROTATIONAL connection:
    If not both components are ROTATIONAL COMPONENTs
        issue an error message and exit the function
    else
        attempt to connect them ( Each component will do error checking and may
        still refuse to connect )
To do a FLOW connection:
    If the component is not a FLOW COMPONENT
        issue an error message and exit the function
    else
        attempt to connect the two objects ( The connection may still be refused by either object )
```

void disconnect_components(void)

This function disconnects two objects from each other. The function's error trapping makes sure that enough objects exist to have a connection in the first place and that the two picked objects are connected. The objects are then disconnected using different methods depending on how they are connected.

void select_all(void)

All of the components and flow stations can be selected for editing simultaneously using this function. This comes in handy when an action should apply to all highlighted components.

void unselect_all(void)

All of the components and flow stations can be deselected simultaneously using this function.

public:

The following constructors are available:

Configuration_Window(CONFIGURATION_WINDOW_DECLARATIONS)

Configuration_Window(CONFIGURATION_WINDOW_DECLARATIONS, char* _title)

Configuration_Window(CONFIGURATION_WINDOW_DECLARATIONS, char* _title, float _character_height)

Configuration_Window(CONFIGURATION_WINDOW_DECLARATIONS, float _thickness, char* _title)

Configuration_Window(CONFIGURATION_WINDOW_DECLARATIONS, float _thickness, char* _title, float _character_height)

int get_NEPP_exit_flag(void)

Return the flag signifying whether a request to exit the configuration has been made.

void set_graphics_info(void)

Create and stock the graphics_info object with the graphics information used by the configuration window.

void turn_ETC_on(void)

Turn on immediate mode graphics.

void turn_ETC_off(void)

Turn off immediate mode graphics.

Formatted_Pop_Up_Menu* pop_up(float _pop_x, float _pop_y, const char* _title, const char* _source)

Pop up a menu formatted according to data from the file, _source. A pointer to the formatted menu is returned for processing purposes.

_pop_x, _pop_y ..... *x and y location for pop-up menu*

_title ..... *Title of pop-up menu*

_source ..... *Name of file containing the menu format*

int process_object(Formatted_Pop_Up_Menu* _pop_up)

Process the formatted pop-up menu for object storage. The choices are "Engine...", "Engine Component..." and "Cancel". "Engine..." pops up the engine object storage menu. "Engine Component..." pops up the engine component storage menu.

_pop_up ..... *Pointer to the pop-up menu object which is to be processed*

int process_engine_object(Formatted_Pop_Up_Menu* _pop_up)

Process the formatted pop-up menu for engine configuration storage. The choices are "Load", "Save" and "Cancel".

_pop_up ..... *Pointer to the pop-up menu object which is to be processed*

int process_component_object(Formatted_Pop_Up_Menu* _pop_up)

Process the formatted pop-up menu for engine component storage. The choices are "Load", "Save", "Create" and "Cancel".

_pop_up ..... *Pointer to the pop-up menu object which is to be processed*

int read_NEPP_input_file(char* _file_name)

This function reads an NEPP input file for an engine configuration. It parses the file and executes all the steps automatically that would have been done manually to create the configuration.

_file_name ..... *Name of NEPP input file*

Class Name: Formatted_Pop_Up_Menu

Description: This class was created using the pop-up menu primitive to facilitate the creation and editing of formatted input menus. The class accepts an array of text strings as arguments. It then parses these strings for format and escape sequences to automatically create a correctly sized menu with labels, text input fields and push buttons. The format strings are processed similar to the way the C library functions *printf* and *scanf* are processed. The following format and escape sequences are currently supported, where *f* refers to a floating point number, *i* refers to an integer, and shadow thickness refers to the thickness of the beveled shadow on push buttons, frames, etc.:

line height:	**%*f*H**	If *f* is positive or zero the line height of the current and subsequent lines will be set to (*f* * character height). If *f* is negative the line height will be set automatically to the height of the tallest object on that line. Defaults: *f* = -1
line spacing:	**%*f*S**	The spacing following the current and subsequent lines will be set to (*f* * character height). Defaults: *f* = .5
"cursor" shift:	**%*f*>**	Shift internal cursor (*f* * shadow thickness) to the right before resuming drawing.

		Defaults: $f = 1$
	"cursor" shift: **%*f*<**	Shift internal cursor (f * shadow thickness) to left before resuming drawing. Defaults: $f = 1$
	text input field:	**%*i*T** A text input field with frame is constructed for a word with *i* letters. The height of the field is 2 * (character height + shadow thickness). Defaults: *i* must be specified.
	Push button: **%*i*B**	A push button is constructed. The push button is sized to fit a label with *i* characters. The height of the push button is 2 * (character height + shadow thickness). Defaults: *i* must be specified.
	Line feed: **/n**	Inserts a line feed with carriage return using the currently set line height and spacing.

Header File Name: formatted_pop_up_menu.h

Derived from: This is a base class.

Member Classes:

private:

Pop_Up_Menu* pop_up ..... *This is the pop-up menu object on which the formatted menu items will be displayed.*

Color_Group* color ..... *This object contains the color information needed by the pop-up menu constructor*

Label** label_item ..... *This is a dynamic array of label objects used for the menu's text. The size of the array is increased as the format array is parsed and more text is encountered.*

Text_Input** text_input_item ..... *This is a dynamic array of text input items used for the menu's input fields. This array also grows dynamically as the format array is parsed.*

Frame** frame_item ..... *Dynamic array of frame items to go along with the text input items. Each text input item is framed by a frame_item.*

Push_Button** button_item ..... *Dynamic array of buttons. This array also grows dynamically as the format array is parsed.*

Member Data:

public:

float x_location ..... *x location of pop-up menu reference point*

float y_location ..... *y location of pop-up menu reference point*

int horizontal_alignment ..... *Horizontal position of reference point on pop-up menu - LEFT, CENTER or RIGHT*

int vertical_alignment ..... *Vertical position of reference point on pop-up menu - TOP, CENTER or BOTTOM*

float menu_width ..... *Width of the pop-up menu*

float menu_height ..... *Height of the pop-up menu*

char* title ..... *Title of the pop-up menu*

float title_height ..... *Character height of the pop-up menu title*

int font ..... *Font used in the pop-up menu*

float thickness ..... *Thickness of bevel shadow used in pop-up menu*

char* line ..... *The line of the format currently being parsed*

char** format_text ..... *Array of text strings containing formats*

char** default_text ..... *Array of text defaults for text input fields and button labels*

float char_height ..... *Height of text in menu (not title)*

float line_spacing_default ..... *Default line spacing*

float current_line_spacing ..... *Currently used line spacing*

int line_height_mode ..... *The line height can be specified manually (0) or it can be set automatically (1) to the height of the tallest item on a line*

float left_margin ..... *Margin from shadow to closest graphic in menu*

float right_margin

float top_margin

float bottom_margin

float* line_height ..... *Dynamic array of line heights for n lines*

float* line_spacing ..... *Dynamic array of line spacings for n lines*

float* y ..... *Dynamic array of vertical line centers*

float x ..... *Current horizontal cursor position*

int number_of_lines ..... *Number of lines in the menu*

float width ..... *Maximum calculated required width*

float height ..... *Maximum calculated required height*

char** label ..... *Dynamic array of label strings*

float* label_x ..... *Dynamic array of label x positions*

float* label_length ..... *Dynamic array of label lengths*

int* label_line ..... *Dynamic array of line numbers of lines to which labels belong*

int label_count ..... *Number of labels counted*
int* text_input_id ..... *Array of event ids for all of the menu items*
int start_id ..... *Number where event ids are allowed to start*
int* text_input_char_num ..... *Dynamic array of numbers of characters of text input fields*
float* text_input_x ..... *Dynamic array of text_input x positions*
char** input_field_text ..... *Dynamic array of strings to initialize text input fields*
float* text_length ..... *Dynamic array of text input text lengths*
int* text_input_line ..... *Dynamic array of line numbers of lines on which text inputs go*
float* frame_width ..... *Dynamic array of frame widths of text input fields*
float* frame_height ..... *Dynamic array of frame heights of text input fields*
int text_input_count ..... *Number of text inputs counted*
int* button_event_id ..... *Dynamic array of push button event ids*
int* button_char_num ..... *Dynamic array of numbers of characters of push button label*
float* button_x ..... *Dynamic array of button x positions (centered)*
char** button_label ..... *Dynamic array of strings to initialize button labels*
float* button_label_length ..... *Dynamic array of lengths of button labels*
int* button_line ..... *Dynamic array of line numbers of lines on which buttons go*
float* button_width ..... *Dynamic array of button widths*
float* button_height ..... *Dynamic array of button heights*
int button_count ..... *Number of buttons counted*

Member Functions:

FORMATTED_POP_UP_MENU_DECLARATIONS is defined as the following list of arguments:
float _x, float _y, int _horizontal_alignment, int _vertical_alignment, const char *_title, float _title_height, char **_format_text, char **_default_text, float _character_height, int _font, float _shadow_thickness, Color_Group *_color

_x ..... *x-position of window*
_y ..... *y-position of window*
_horizontal_alignment ..... *horizontal reference position on window (LEFT, CENTER, or RIGHT)*
_vertical_alignment ..... *vertical reference position on window (TOP, CENTER, or BOTTOM)*
_title ..... *Title of the menu*
_title_height ..... *Character height of the menu title*
_format ..... *Array of format strings*
_default_text ..... *Array of default text inputs and labels*
_character_height ..... *Character height for the main portion of the menu*
_font ..... *Font for the menu*
_shadow_thickness ..... *Thickness of the bevel shadow of the menu*
_color ..... *Object containing color information for the menu*

private:
float get_title_length(void)
This function returns the length in modeling coordinates of the title string. This is needed to determine the width of the menu.

void initialize_line(int _line_number)
This function copies the specified string from the format array into the parsing string, line. It also initializes the cursor's x position to zero, sets the line height to zero and sets the line spacing to default.
_line_number.....The number of the current menu line being parsed.
void process_cursor_shift_right(void)
Parse the format line for a ">" format. If it is found in the first position, add the correct width to the cursor x location and remove the format from the format line.
void process_cursor_shift_left(void)
Parse the format line for a "<" format. If it is found in the first position, subtract the correct width from the cursor x location and remove the format from the format line.
void process_line_height_format(int _line_number)
Parse the format line for an "H" format. If it is found in the first position, then if the line height format is negative set the line height mode to automatic. Else set the mode to manual and set the line_height to the specified format. Remove the format from the format line.
_line_number.....The number of the current menu line being parsed.
void process_line_spacing_format(int _line_number)
Parse the format line for a "S" format. If it is found in the first position, then set the line spacing to the specified size. Remove the format from the format line.
_line_number.....The number of the current menu line being parsed.
void process_text_input_format(int _line_number)
Parse the format line for a "T" format. If it is found in the first position,
- allocate memory for a new text input item,
- calculate size and position of the input field,
- if the line height mode is automatic calculate the new line height,
- calculate the new width of the menu, and
- remove the format from the format line.

_line_number.....The number of the current menu line being parsed.
void process_button_format(int _line_number)
Parse the format line for a "B" format. If it is found in the first position,
- allocate memory for a new push button,
- calculate size and position of the button,
- if the line height mode is automatic calculate the new line height,
- calculate the new width of the menu, and
- remove the format from the format line.

_line_number.....The number of the current menu line being parsed.
void process_label(int _line_number)
Parse the format line for any format. If none is found in the first position,
- allocate memory for a new label,
- calculate size and position of the label,
- if the line height mode is automatic calculate the new line height,
- calculate the new width of the menu, and
- remove the label from the format line.

_line_number.....The number of the current menu line being parsed.
void terminate_line(int _line_number)
Set the y position of the line to the center of the line using information about the line height and spacing. Also clean up by deleting the format line from memory.
_line_number.....The number of the current menu line being parsed.
void process_line(int _line_number)
Parses the format line for all supported formats and escape codes.
_line_number.....The number of the current menu line being parsed.
void create_pop_up_menu(void)

Process the format text array for labels, text inputs and buttons. Find their dimensions to calculate the full height and width of the menu. Create the pop-up menu.

void create_menu_items(void)

Create and initialize the menu items and set their attributes.

void add_menu_items(void)

Add all of the menu items to the pop-up menu.

public:

There is only one constructor for the Formatted_Pop_Up_Menu class:

Formatted_Pop_Up_Menu(FORMATTED_POP_UP_MENU_DECLARATIONS)

The constructor function initializes all of the necessary variables, creates the menu and menu items and adds the menu items to the menu.

int get_text_input_count(void)

Return the number of text input fields in the menu.

int get_button_count(void)

Return the number of push buttons in the menu.

Text_Input **get_text_input_items(void)

Return an array of pointers to text input objects. This function is used in the external processing loop.

Push_Button **get_button_items(void)

Return an array of pointers to push button objects. This function is used in the external processing loop.

int get_number_of_lines(void)

This function parses the array of format strings for '/n' escape codes. It splits the line at each such escape code and finally returns the total number of lines.

float get_left_bound(void)

Calculate and return the left-most usable position. This means inside the border and left margin of the menu.

float get_right_bound(void)

Calculate and return the right-most usable position. This means inside the border and right margin of the menu.

float get_top_bound(void)

Calculate and return the top-most usable position. This means inside the border and top margin of the menu.

float get_bottom_bound(void)

Calculate and return the bottom-most usable position. This means inside the border and bottom margin of the menu.

int *get_text_input_id(void)

Return an array of event ids for the text input fields.

int *get_button_id(void)

Return an array of event ids for the push buttons.

int get_ok_id(void)

This function is one of convenience and only applies to menus in which the second to last button is an "Ok" button. It returns the event id of the second to last push button.

int get_cancel_id(void)

This function is one of convenience and only applies to menus in which the last button is a "Cancel" button. It returns the event id of the last push button.

void turn_ETC_on(void)

Turn immediate mode graphics on.

Event * await_event(Interface_Manager *_iman)

Returns an event from the interface manager.

_iman.....Pointer to the interface manager object

Class Name: Engine_Component

Class Description: This class holds all the information and methods needed to describe an NEPP engine component. The class can be divided into the following categories

- Linked list data and functions
- Graphics related data
- Icon related data and functions
- Data input related data and functions
- Engine component configuration data and functions

Header file Name: engine_component.h

Derived from: This is a base class.

Member Classes:

Graphics_Info graphics_info*.....*Pointer to object containing relevant graphics information*

Engine_Component_Icon icon*.....*The engine component's icon representation object*

*const Connection** connection_list_ptr*.....*This is a pointer to the linked list of connections maintained in the configuration window object. Using a pointer ensures that the actual connection list is always accessible and up-to-date even if changes to it occur outside this class.*

protected:

Engine_Component* previous.....*Pointer to the previous item in the linked list*

Engine_Component* next.....*Pointer to the next item in the linked list*

Member Data:

protected:

char* filename.....*A string containing the path and root of the filename used for all configuration files*

int first_text_input_index.....*This is the index of the first text input entry stored in the saved_input array.*

int first_button_index ..... *This is the index of the first button label stored in the saved_input array.*

int name_index ..... *Index of the engine component name stored in the saved_input array*

int* upstream_index ..... *Indices of the upstream connection names stored in the saved_input array*

int* downstream_index ..... *Indices of the downstream connection names stored in the saved_input array*

int* connection_index ..... *Indices of the rotational connection names stored in the saved_input array*

int ok_index ..... *Index of the Ok button*

int cancel_index ..... *Index of the Cancel button*

char* name ..... *Engine component name as supplied by the user*

Component_Type type ..... *This is the component type. It does not correspond to the component types used by NEPP! The possible engine component types are:*
- *NO_TYPE*
- *INLET_TYPE*
- *DUCT_TYPE*
- *BURNER_TYPE*
- *GAS_GENERATOR_TYPE*
- *WATER_INJECTOR_TYPE*
- *FAN_TYPE*
- *COMPRESSOR_TYPE*
- *TURBINE_TYPE*
- *HEAT_EXCHANGER_TYPE*
- *FLOW_SPLITTER_TYPE*
- *FLOW_MIXER_TYPE*
- *EJECTOR_TYPE*
- *NOZZLE_TYPE*
- *LOAD_TYPE*
- *PROPELLER_TYPE*
- *SHAFT_TYPE*

int number ..... *Component number used by NEPP*

char* blank_var ..... *This is a pointer to blank string. It is used to initialize name pointers rather than initializing them to NULL.*

int max_upstream_connections ..... *Maximum number of upstream connections this component is allowed to have*

int max_downstream_connections ..... *Maximum number of downstream connections this component is allowed to have*

int max_rotational_connections ..... *Maximum number of rotational connections this component is allowed to have*

const char* const** local_upstream_names ..... *This is an array of pointers to the names of the upstream connections. Using pointers to names ensures that the names accessed by this component are always up-to-date even if they are changed outside this object.*

const char* const** local_downstream_names ..... *This is an array of pointers to the names of the downstream connections. Using pointers to names ensures that the names accessed by this component*

are always up-to-date even if they are changed outside this object.

const char* const** local_rotational_names ..... *This is an array of pointers to the names of the rotational connections. Using pointers to names ensures that the names accessed by this component are always up-to-date even if they are changed outside this object.*

public:

char** saved_input ..... *This is an array of strings storing the input information displayed by the input menu. It contains text field entries and button labels.*

float input_data[15] ..... *This array of numbers is equivalent to the KONFIG array used by NEPP. It contains the physical definition of the component.*

Member Functions:

ENGINE_COMP_DECLARATIONS is defined by the following list of arguments:

Graphics_Info *_graphics_info, const Connection * *_connection_list_ptr, float _x, float _y

_graphics_info ..... *Object containing relevant graphics information*

_connection_list ..... *Pointer to the linked list of connections*

_x, _y ..... *Location for the engine component icon*

protected virtual:

void verify_saved_input(void)

Makes sure all the connections specified in the saved_input array are still connected. If they are not, like for example in the event that a component or connection was deleted, the corresponding saved_input element is set to "".

void process_push_buttons(Event* _event)

This function handles push button events. In the engine component input menus push buttons usually have toggling labels. This function takes care of relabeling the buttons.

_event ..... *Object containing event information*

int verify_input(Text_Input** _text_input, Push_Button** _button)

This function checks the current input against invalid values. Valid inputs are set and invalid inputs are set to valid defaults. This function returns REFUSED if invalid values were encountered. The following pseudo code explains the structure of this function:

Attempt to change the name of the component to the name in the input menu.
Attempt to connect the desired connections
    Loop through the rotational connection indices
        First make sure there is nothing connected in the current location
        If there is a rotational connection in the current location
            Store the name of the connected component
            Disconnect the two components by deleting the connection
        Get the name of the new component from the input field
        If the name is blank then set the return code to 0
        else find the corresponding component
        If the component is this component or NULL set the return code to REFUSED
        If the component is not a rotational component set the return code to REFUSED
        If the component is already connected through one of the rotational connections disconnect it.
        Create a rotational connection and associate both components with it.
            Create a linked list connection

Set the name to the next available NEGATIVE integer in the list of connections
If the new_connection is refused by either component
disconnect any successful connection and
reconnect the previously connected component if there was one.
Loop through the desired upstream connection input fields
If the desired connection is not allowed set the return code to REFUSED
Loop through the desired downstream connection input fields
If the desired connection is not allowed set the return code to REFUSED
Return the return code whether it be REFUSED or not
_text_input.....*The array of text input objects*
_button.....*The array of push button objects*

public virtual:

void set_type(void)

The **set_type** function without argument pops up the type input menu (**pop_up_type_menu**)

void set_type(Component_Type _type)

The set_type function with argument sets the engine component type. Note that this function only works in the base class since the derived classes' types are defined by the class itself.

_type.....*Engine component type*

private:

void pop_up_type_menu(void)

This function pops up and processes a menu allowing the user to choose a component type from a list of radio buttons labeled with each component type. Upon exiting it sets the type variable to the type indicated by the depressed radio button.

void pop_input_data_menu(void)

This function reads the Formatted_Pop_Up_Menu format file for the component and creates the input menu. It then calls the processing function **process_input_data_menu.**

void process_input_data_menu(Formatted_Pop_Up_Menu* _menu)

Processes the input pop-up menu. The push buttons receive special attention since they must be toggled by the component. The input field entries are only checked for validity when the "Ok" button is clicked. While invalid field entries are encountered the menu cannot be exited with the "Ok" button. The "Cancel" button exits without further processing.

_menu.....*This is the input data menu object which is to be processed in this function*

protected:

void set_previous(Engine_Component* _previous)

Set this element's previous pointer to point to _previous.

_previous.....*A pointer to the previous element in the linked list of engine components*

void set_next(Engine_Component* _next)

Set this element's next pointer to point to _next.

_next.....*A pointer to the next element in the linked list of engine components*

public:

There is only one available constructor for the Engine_Component class:

Engine_Component(ENGINE_COMP_DECLARATIONS)

Create the component icon and initialize all other engine component data.

int count_elements(void)

Return the number of linked list elements in this list.

Engine_Component* get_first(void)

Return the first linked list element in this list.

Engine_Component* get_last(void)
Return the last linked list element in this list.
Engine_Component* get_previous(void)
Return the linked list element pointed to by the pointer to previous.
Engine_Component* get_next(void)
Return the linked list element pointed to by the pointer to next.
void add_element(Engine_Component* _new_element)
Attach the element, _new_element, at the end of the list.
_new_element ..... *A pointer to the engine component object to be added to the linked list*
void insert_element(Engine_Component* _new_element)
Insert the element, _new_element, after this element and before the element pointed to by the pointer to next.
_new_element ..... *A pointer to the engine component object to be inserted right after the current component in the linked list*
void remove_this_element(void)
Delete this element. The destructor will take care of first bypassing this element so that the linked list stays intact.
void remove_this_element(Engine_Component* _this_element)
Delete the element, _this_element. The destructor will take care of first bypassing the element so that the linked list stays intact.
_this_element ..... *Pointer to the element to be removed from the linked list*
void bypass_this_element(void)
This element of the linked list can separate itself from the linked list by calling this function. The linked list is repaired by adjusting the next pointer of the previous element and the previous pointer of the next element to point to each other.
void set_defaults(void)
Read defaults into the input data array. The defaults are stored in the components' defaults file with the extension ".defaults".
int change_name(char* _name)
This function sets the name after error trapping. The error trapping ensures that the new name is a valid name and does not already exist.
_name ..... *The proposed name string for the component*
void update_input_data_menu(Formatted_Pop_Up_Menu* _menu)
This function updates the text input fields and button labels according to data in the saved_input array. This function can be used to reset the input menu to a saved state.
_menu ..... *The data input menu object whose menu items are to be updated*
void move_selected_components(float _delta_x, float _delta_y)
This function is responsible for moving the component icons on the display. It traverses the linked list of engine components and requests the move from each selected icon.
_delta_x, _delta_y ..... *x and y translations to be applied to all selected engine component icons*
int count_selected_components(void)
This function traverses the linked list of engine components and counts the components with icons in the selected state.
int read_data_from_file(FILE *_stream)
This function reads the NEPP data from a file into the input_data array.
_stream ..... *Pointer into the data file where the data list begins*

void set_name(void)
The **set_name** function without arguments pops up the component's data input menu.
void set_name(char *_name)
The **set_name** function with argument sets the name without error trapping.
_name.....*New component name*
void set_number(int _number)
Set the component number used by NEPP.
_number.....*New component number (corresponds to NEPP component numbering scheme)*
const char* get_name(void)
Return the component's name.
const char* const* get_name_ptr(void)
Return the location of the pointer to the component name in memory. This allows objects outside this object to have an updated view of the component name regardless of any changes to it.
Component_Type get_type(void)
Return the component type.
int get_number(void)
Return the component number as used by NEPP.
Engine_Component* get_component(int _number)
Return the engine component with a specified NEPP number.
_number.....*Current component number (corresponds to the NEPP numbering scheme)*
Engine_Component* PHIGS_get_component(Pint _structure_id)
Return the engine component whose icon corresponds to a specified PHIGS structure id.
_structure_id.....*PHIGS structure id*
Engine_Component* get_component(const char* _name)
Return the engine component with a specified name.
_name.....*Component name*
int can_connect_upstream(Flow_Connection* _flow_station)
Do preliminary error trapping to determine whether this component is allowed to connect to a specified upstream flowstation.
_flow_station.....*Flowstation object*
int can_connect_downstream(Flow_Connection* _flow_station)
Do preliminary error trapping to determine whether this component is allowed to connect to a specified downstream flowstation.
_flow_station.....*Flowstation object*
int can_connect_rotational(Rotational_Connection* _connection)
Do preliminary error trapping to determine whether this component is allowed to connect to a specified rotational connection.
_connection.....*Rotational connection object*
int connect_upstream(Flow_Connection* _flow_station)
This function finds the next available slot index to do an upstream connection and then calls the overloaded function **connect_upstream(_index, _name)** to do the final error trapping and connection.
_flow_station.....*Flowstation object*
int connect_downstream(Flow_Connection* _flow_station)
This function finds the next available slot index to do an downstream connection and then calls the overloaded function **connect_downstream(_index, _name)** to do the final error trapping and connection.
_flow_station.....*Flowstation object*
int connect_rotational(Rotational_Connection* _connection)

This function finds the next available slot index to do a rotational connection and then calls the overloaded function **connect_rotational(_index, _name)** to do the final error trapping and connection.

_connection ..... *Rotational connection object*

int connect_upstream(int _index, const char* _name)

This function takes care of the final error trapping and connecting the component to the upstream flowstation. The following pseudo code explains the logic:

If string is blank, bypass all this:
Extract integer from _name
If no integer was found return REFUSED
Convert the integer string to an integer and store
Check if connection _name already exists and store it.
If it doesn't return REFUSED.
Remember current entry at _index in the array of upstream name pointers
then disconnect it.
If _name is blank return 0 at this point
Find any other entry with the same name both in the upstream and downstream name pointer arrays, store the index, and
disconnect it.
Check if upstream connection is allowed by this component
Check if downstream connection is allowed by the flow connection
If connection was not refused by this component or by flow station
set the entry at location _index in the local array of upstream names, to the pointer to the name of the new flow_station.
Connect this component to the new flow station.
else
Set the name at the position _index back to what it was.
Return _name back to the position where it was before.

_index ..... *Index into the local_upstream_names array where the new connection should be placed*

_name ..... *Name of the new flowstation*

int connect_downstream(int _index, const char* _name)

This function takes care of the final error trapping and connecting the component to the downstream flowstation. The logic is similar to that used in the **connect_upstream(_index, _name)** function.:

_index ..... *Index into the local_downstream_names array where the new connection should be placed*

_name ..... *Name of the new flowstation*

int connect_rotational(int _index, const char* _name)

This function takes care of the final error trapping and connecting the component to the rotational connection. The logic is similar to that used in the **connect_upstream(_index, _name)** function.:

_index ..... *Index into the local_rotational_names array where the new connection should be placed*

_name ..... *Name of the new rotational connection*

int is_connected_to(Engine_Component* _component)

Determine if this engine component is connected to a specified component via upstream or downstream flowstation or rotational connection.

_component ..... *Engine component object*

int count_local_connections(void)

Return number of upstream, downstream and rotational connections.

void disconnect(Engine_Component* _component)

Undo any rotational connection between this component and the specified component by deleting the rotational connection..

_component .....*Engine component object*

void disconnect(Connection* _connection)

Undo the specified connection, whether it be upstream, downstream or rotational. The connection objects themselves stay intact but they are separated from this component. Rotational connections should always be deleted rather than just disconnected!

_connection .....*Flow OR rotational connection object*

void disconnect_all(void)

Undo all connections to this component. Rotational Connections are deleted.

Class Name: Connection

Class Description: This is the base class for NEPP connections. It contains:

- Doubly linked list data and methods
- Icon related data and functions
- General connection related data and functions

Header file Name: connection.h

Derived from: This is a base class.

Member Classes:

Graphics_Info* graphics_info ..... *Object containing relevant graphics information*

Connection* previous ..... *Pointer to the previous connection element in the linked list*

Connection* next ..... *Pointer to the next connection element in the linked list*

Icon* icon ..... *Icon object containing the graphical representation of the connection*

const Engine_Component** component_list_ptr ..... *Pointer to the linked list of engine components*

Member Data:

private:

int number ..... *Flow station number used by NEPP*

char* name ..... *Connection name as supplied by the user*

protected:

char* blank_var ..... *Pointer to a memory location containing a blank string for initializing name pointers*

Member Functions:

CONNECTION_DECLARATIONS is defined by the following argument list:

Graphics_Info* _graphics_info, const Engine_Component** _component_list_ptr, float _x, float _y

_graphics_info ..... *Object containing relevant graphics information*

_component_list_ptr ..... *Pointer to linked list of engine component objects*

_x, _y ..... *Location for connection icon*

public virtual:

int is_connected_to(Engine_Component* _component)

Determine whether or not this connection is connected to a specified engine component.

_component ..... *Engine component object*

int count_local_component(void)

Return the number of engine components connected to this connection.

void disconnect(Engine_Component* _component)

Remove a specified component name pointer from the array of connected engine component name pointers.

_component ..... *Engine component to be disconnected*

protected:

void set_previous(Connection* _previous)

Set the pointer to the previous linked list element to a specified connection object.

_previous ..... *Pointer to connection object*

void set_next(Connection* _next)

Set the pointer to the next linked list element to a specified connection object.

_next ..... *Pointer to connection object*

public:

int count_elements(void)

Return the number of connection elements in the linked list of connections.

Connection * get_first(void)

Return a pointer to the first element in the linked list.

Connection * get_last(void)

Return a pointer to the last element in the linked list.

Connection * get_previous(void)

Return a pointer to the previous element in the linked list.

Connection * get_next(void)

Return a pointer to the next element in the linked list.

void add_element(Connection *_new_connection)

Add a new element at the end of the linked list.

_new_connection ..... *Connection object to be appended to the linked list*

void insert_element(Connection *_new_connection)

Insert a new element into the linked list right after the current one.

_new_connection ..... *Connection object to be inserted to the linked list*

void remove_this_element(void)

Delete the current connection element. Connections to components are cleaned up in the destructor.

void remove_this_element(Connection* _this_connection)

Delete the specified element. Connections to components are cleaned up in the destructor.

_this_connection ..... *Connection object to be deleted*

void bypass_this_element(void)

Separate the element from the linked list. The linked list remains intact after the next pointer of the previous element and the previous pointer of the next element were reassigned.

void move_selected_connections(float _delta_x, float _delta_y)

This function traverses all connections and moves selected icons by the specified amount.

_delta_x, _delta_y ..... *Amount by which to move all selected connections*

int count_selected_connections(void)

This function returns the number of selected connections.

void set_number(int _number)

Set the number of the connection.

_number ..... *New number for connection*

int get_number(void)

Return the number of the connection.

const char * get_name(void)

Return the name of the connection.

const char * const * get_name_ptr(void)
Return the pointer to the name of the connection. This allows objects outside this class to have up-to-date access to the name of this object.
Connection *get_connection(int _number)
Return the connection with the specified number.
_number ..... *Number of connection*
Connection *get_connection(const char *_name)
Return the connection with the specified name.
_name ..... *Name of connection*
Connection *PHIGS_get_connection(Pint _structure_id)
Return the connection whose icon has the specified PHIGS structure id.
_structure_id ..... *PHIGS structure id*

Class Name: Flow_Connection

Class Description: The flow connection serves as the NEPP flow station. It is used to connect two or more flow components. The Flow_Connection contains:

- Data input related functions
- Flow connection specific data and functions

Header file Name: flow_connection.h

Derived from: Connection

Member Classes: none

Member Data:

protected:

char** saved_input ..... *Array of data input strings from the data input menu*

int max_upstream_components ..... *Number of maximum allowable upstream components*

int max_downstream_components ..... *Number of maximum allowable downstream components*

public:

const char* const** local_upstream_names ..... *Array of pointers to names of upstream connected components*

const char* const** local_downstream_names ..... *Array of pointers to names of downstream connected components*

Member Functions:

FLOW_CONNECTION_DECLARATIONS is defined by the following list of arguments:

Graphics_Info *_graphics_info, const Engine_Component * *_component_list_ptr, float _x, float _y

_graphics_info ..... *Object containing relevant graphics information*

_component_list_ptr ..... *Pointer to the linked list of engine components*

_x, _y ..... *Location for the flow connection icon*

protected:

void pop_input_data_menu(void)

This function creates a format array and creates the input menu from it. The menu consists of one input field with the flowstation name and a list of the connected upstream and downstream components. It then calls the processing function **process_input_data_menu.**

void process_input_data_menu(Formatted_Pop_Up_Menu* _menu)

Process the input pop-up menu. The name field is only checked for validity when the "Ok" button is clicked. While invalid field entries are encountered the menu cannot be exited with the "Ok" button. The "Cancel" button exits without further processing.

_menu ..... *This is the input data menu object which is to be processed in this function*

int verify_input(Text_Input** _text_input_item, Push_Button** _push_button_item)

This function checks the current input against invalid values. If the name field does not contain an integer or if this integer already exists as a connection name, then this function returns REFUSED.

_text_input_item ..... *The array of text input objects*

_push_button_item ..... *The array of push button objects*

int change_name(char* _name)

This function sets the name after error trapping. The error trapping ensures that the new name is a valid name and does not already exist.

_name ..... *The proposed name string for the connection*

void update_input_data_menu(Formatted_Pop_Up_Menu* _menu)

This function updates the text input fields and button labels according to data in the saved_input array. This function can be used to reset the input menu to a saved state.

_menu ..... *The data input menu object whose menu items are to be updated*

public:

Flow_Connection(FLOW_CONNECTION_DECLARATIONS)

The constructor creates the flow connection icon and initializes some connection data.

void get_input(void)

Pops up the input menu and sets the name of the connection according to the input

void set_number(int _number)

Set the NEPP number of the flowstation

_number ..... *Connection number as used by NEPP*

int can_connect_upstream(Engine_Component* _component)

Determines whether a specified engine component is allowed to connect upstream of this connection.

_component ..... *Engine component object to be connected*

int connect_upstream(Engine_Component* _component)

If it is not already there, include the name pointer of the specified engine component in the list of upstream component name pointers.

_component ..... *Engine component object to be connected*

int can_connect_downstream(Engine_Component* _component)

Determines whether a specified engine component is allowed to connect downstream of this connection.

_component ..... *Engine component object to be connected*

int connect_downstream(Engine_Component* _component)

If it is not already there, include the name pointer of the specified engine component in the list of downstream component name pointers.

_component ..... *Engine component object to be connected*

int count_local_upstream_components(void)

Return the number of name pointers in the array of upstream component name pointers.

int count_local_downstream_components(void)

Return the number of name pointers in the array of downstream component name pointers.

int is_connected_to(Engine_Component* _component)

Determine whether a specified component is connected upstream or downstream of the flowstation by scanning the arrays of connected component name pointers.

_component .....*Engine component object*

int count_local_components(void)

Return the number of name pointers in both the upstream and downstream component name pointer arrays.

void disconnect(Engine_Component* _component)

Remove the component's name pointer from the upstream and downstream array and remove the graphical connection arrow from the icon.

_component .....*Engine component object to be removed*

Class Name: Rotational_Connection

Class Description: The rotational connection serves to make connections between rotational engine components. Currently NEPP only allows connections between a rotational component and a shaft component. Each rotational connection must have exactly two connected components, no more. Since there is no data associated with the rotational connection there is no data input menu. As far as the user knows the connection does not even have a name. For consistency in methods, however, an internal name is used. The internal name is set to be a negative, non-zero integer.

Header file Name: rotational_connection.h

Derived from: Connection

Member Classes: none

Member Data:

protected:

char** saved_input ..... *Array of data input strings from the data input menu*

int max_components ..... *Number of maximum allowable components*

public:

const char* const** local_component_names ..... *Array of pointers to names of connected rotational components*

Member Functions:

ROTATIONAL_CONNECTION_DECLARATIONS is defined by the following list of arguments:

Graphics_Info *_graphics_info, const Engine_Component * *_component_list_ptr, float _x, float _y

_graphics_info ..... *Object containing relevant graphics information*

_component_list_ptr ..... *Pointer to the linked list of engine components*

_x, _y ..... *Location for the flow connection icon*

public:

Rotational_Connection(ROTATIONAL_CONNECTION_DECLARATIONS)
The constructor creates the rotational connection icon and initializes some connection data.
int can_connect(Engine_Component* _component)
Determines whether a specified engine component is allowed to connect to this connection.
_component .....*Engine component object to be connected*
int connect(Engine_Component* _component)
If it is not already there, include the name pointer of the specified engine component in the list of rotational component name pointers. Add a line from the connection coordinates to the engine component icon coordinates to the rotational connection icon.
_component .....*Engine component object to be connected*
int is_connected_to(Engine_Component* _component)
Determine whether a specified component is connected to the rotational connection by scanning the array of connected component name pointers.
_component .....*Engine component object*
int count_local_components(void)
Return the number of name pointers in the rotational component name pointer array.
void disconnect(Engine_Component* _component)
Remove the component's name pointer from the rotational name pointer array and remove the graphical connection line from the icon.
_component .....*Engine component object to be removed*

Class Name: Icon

Class Description: This class is the base class for all of the graphical representations used in this program. Engine component icon, flow connection icon and rotational connection icon are derived from it. The base class provides the means to draw and manipulate a named PHIGS graphics structure. Manipulations include translation, selection, activation. Some or all of these functions should be redefined in the derived class to suit the nature of the particular icon.

Header file Name: icon.h

Derived from: This is a base class.

Member Classes:

PHIGS_Structure_ID structure ..... *Icon's PHIGS structure number*
PHIGS_View* view ..... *PHIGS view number*
Color_Group* color ..... *Color information object*
Color_Group* saved_color ..... *Saved color to revert to if necessary*

Member Data:

int state ..... *State of the icon: ACTIVE or INACTIVE*
int select_state ..... *State of the icon: SELECTED or UNSELECTED*
float x, y ..... *Location of upper left hand corner of the icon*
float x_center, y_center ..... *Location of the center of the icon*
float width, height ..... *Dimensions of the icon*
float shadow_thickness ..... *Thickness of the bevel shadow of the icon's slab (if it has a slab)*
int ETC_flag ..... *Immediate mode graphics flag: ON or OFF*
char* name ..... *Name of the icon: This is used for a label*
float character_height ..... *Height of the label in modeling coordinates*

Member Functions:

ICON_DECLARATIONS is defined as the following list of arguments:
float _x, float _y, int _horizontal_alignment, int _vertical_alignment, float _width, float _height

_x ..... *x location of icon's reference point*
_y ..... *y location of icon's reference point*
_horizontal_alignment ..... *Position of reference point on icon: LEFT, CENTER, or RIGHT*
_vertical_alignment ..... *Position of reference point on icon: TOP, CENTER, or BOTTOM*
_width ..... *Width of icon in modeling coordinates*
_height ..... *Height of icon in modeling coordinates*

protected virtual:

void initializer(ICON_DECLARATIONS, float _shadow_thickness, Color_Group* _color)

Initialize icon data.

_shadow_thickness ..... *Thickness of bevel shadow*
_color ..... *Color information object*

void create_structure(void)

Open a PHIGS structure. Call the functions to create the pieces of the icon. Close the structure.

void display_change(int _perform_flag)

Erases old PHIGS structure and redraws it with the new coordinates and attributes, etc.

_perform_flag ..... *Update display? PERFORM or DO_NOT PERFORM*

public virtual:

void activate(int _perform_flag)

Activate icon. This function sets the state flag to ACTIVE. It should be redefined in the derived class if necessary.

_perform_flag ..... *Update display? PERFORM or DO_NOT_PERFORM*

void deactivate(int _perform_flag)

Deactivate icon. This function sets the state flag to INACTIVE. It should be redefined in the derived class if necessary.

_perform_flag ..... *Update display? PERFORM or DO_NOT_PERFORM*

public:

Icon(ICON_DECLARATIONS)
Icon(ICON_DECLARATIONS, Color_Group* _color)
Icon(ICON_DECLARATIONS, float _shadow_thickness)
Icon(ICON_DECLARATIONS, float _shadow_thickness, Color_Group* _color)
Icon(ICON_DECLARATIONS, char* _name, float _character_height)
Icon(ICON_DECLARATIONS, float _shadow_thickness, Color_Group* _color, char* _name, float _character_height)

_color ..... *Color information object*
_shadow_thickness ..... *Thickness of bevel shadow*
_name ..... *Name of icon to be used for the label*
_character_height ..... *Height of the label text in modeling coordinates*

void associate(PHIGS_View* _view, float _priority)

Associate the icon structure with the PHIGS view with a specified priority.

_view ..... *Object containing PHIGS view information*
_priority ..... *Display priority of this structure with respect to other structures displayed in this view.*

void disassociate(void)

Disassociate the icon structure from the PHIGS view.

int get_structure_id(void)

Return the PHIGS structure id of the icon.

int get_state(void)
Return the state of the icon: ACTIVE or INACTIVE.
int get_select_state(void)
Return the state of the icon: SELECTED or UNSELECTED.
float get_width(void)
Return the width of the icon in modeling coordinates.
float get_height(void)
Return the height of the icon in modeling coordinates.
const float* get_width_ptr(void)
Return a pointer to the icon's width variable. This way objects outside this class can have access to an up-to-date width without inquiring it everytime even if it changes inside this class.
const float* get_height_ptr(void)
Return a pointer to the icon's height variable. This way objects outside this class can have access to an up-to-date height without inquiring it everytime even if it changes inside this class.
float get_x_min(void)
Return minimum x coordinate of icon.
float get_y_min(void)
Return minimum y coordinate of icon.
float get_x_max(void)
Return maximum x coordinate of icon.
float get_y_max(void)
Return maximum y coordinate of icon.
float get_x_center(void)
Return x center coordinate of icon.
float get_y_center(void)
Return y center coordinate of icon.
const float* get_x_ptr(void)
Return a pointer to the icon's x variable. This way objects outside this class can have access to an up-to-date x without inquiring it everytime even if it changes inside this class.
const float* get_y_ptr(void)
Return a pointer to the icon's y variable. This way objects outside this class can have access to an up-to-date y without inquiring it everytime even if it changes inside this class.
void select(int _perform_flag)
Select the icon. This means setting the select_state flag to SELECTED and redrawing the icon in a highlighting color.
_perform_flag ..... *Update display? PERFORM or DO_NOT_PERFORM*
void unselect(int _perform_flag)
Deselect the icon. This means setting the select_state flag to UNSELECTED and redrawing the icon in its regular color.
_perform_flag ..... *Update display? PERFORM or DO_NOT_PERFORM*
void select_toggle(int _perform_flag)
If the icon is selected then deselect it. Else select it.
_perform_flag ..... *Update display? PERFORM or DO_NOT_PERFORM*
void set_color(Color_Group* _color)
Redraw the icon in the specified color scheme.
_color ..... *Color information object*
void move_to(float _x, float _y, int _h_align, int _v_align, int _perform_flag)

Translate the icon to the specified x and y coordinates, aligning the point with the specified reference point on the icon.

_x, _y ..... *Coordinates of the new icon location*
_h_align ..... *Location of reference point on icon: LEFT, CENTER, or RIGHT*
_v_align ..... *Location of reference point on icon: TOP, CENTER, or BOTTOM*
_perform_flag ..... *Update display? PERFORM or DO_NOT_PERFORM*

void move_to_x(float _x, int _h_align, int _perform_flag)

Translate the icon to the specified x coordinates, aligning the point with the specified reference point on the icon.

_x ..... *x - coordinate of the new icon location*
_h_align ..... *Location of reference point on icon: LEFT, CENTER, or RIGHT*
_perform_flag ..... *Update display? PERFORM or DO_NOT_PERFORM*

void move_to_y(float _y, int _v_align, int _perform_flag)

Translate the icon to the specified y coordinates, aligning the point with the specified reference point on the icon.

_y ..... *y - coordinate of the new icon location*
_v_align ..... *Location of reference point on icon: TOP, CENTER, or BOTTOM*
_perform_flag ..... *Update display? PERFORM or DO_NOT_PERFORM*

void move_delta(float _delta_x, float _delta_y, int _perform_flag)

Translate icon by the specified changes in coordinates.

_delta_x ..... *x change in location*
_delta_y ..... *y change in location*
_perform_flag ..... *Update display? PERFORM or DO_NOT_PERFORM*

void move_delta_x(float _delta_x, int _perform_flag)

Translate icon by the specified change in the x-coordinate.

_delta_x ..... *x change in location*
_perform_flag ..... *Update display? PERFORM or DO_NOT_PERFORM*

void move_delta_y(float _delta_y, int _perform_flag)

Translate icon by the specified change in the y-coordinate.

_delta_y ..... *y change in location*
_perform_flag ..... *Update display? PERFORM or DO_NOT_PERFORM*

void set_name(const char* _name, float _character_height)

Set the name variable and redraw icon with the new name as a label.

_name ..... *New icon name*
_character_height ..... *Height of the label text in modeling coordinates*

void turn_ETC_on(void)

Turn immediate mode graphics on.

void turn_ETC_off(void)

Turn immediate mode graphics off.

void ETC_display(void)
This function duplicates the **display_change** function in immediate mode graphics.
int get_ETC_flag(void)
Return the immediate mode graphics flag.

Class Name: Engine_Component_Icon

Class Description: This class contains information and methods specific to the icon used by engine components. The icon is a labeled push button. It can be translated, selected (highlighted) and activated (raised/lowered). The label consists of the icon name. The icon is sized to fit around the label. Changing the icon name also changes the label.

Header file Name: engine_component_icon.h

Derived from: Icon

Member Classes:

Member Data:

Member Functions:

ENGINE_COMPONENT_ICON_DECLARATIONS is defined as the following list of arguments:
float _x, float _y, int _horizontal_alignment, int _vertical_alignment, float _width, float _height

_x ..... *x location of icon's reference point*
_y ..... *y location of icon's reference point*
_horizontal_alignment ..... *Position of reference point on icon: LEFT, CENTER, or RIGHT*
_vertical_alignment ..... *Position of reference point on icon: TOP, CENTER, or BOTTOM*
_width ..... *Width of icon in modeling coordinates*
_height ..... *Height of icon in modeling coordinates*

private:

void create_structure(void)
Open a PHIGS structure. Call the functions to create the pieces of the icon. Close the structure.
void create_name(void)
Set text attributes and execute a PHIGS text primitive with the icon name as argument.
void create_lower_right_bevel(void)
Set polygon attributes and execute a PHIGS polygon primitive for the lower and right bevel of the push button slab.

void create_upper_left_bevel(void)
Set polygon attributes and execute a PHIGS polygon primitive for the upper and left bevel of the push button slab.
void create_face(void)
Set polygon attributes and execute a PHIGS polygon primitive for the face of the push button slab.
public:
Engine_Component_Icon(ENGINE_COMPONENT_ICON_DECLARATIONS)
Engine_Component_Icon(ENGINE_COMPONENT_ICON_DECLARATIONS, Color_Group* _color)
Engine_Component_Icon(ENGINE_COMPONENT_ICON_DECLARATIONS, float _shadow_thickness)
Engine_Component_Icon(ENGINE_COMPONENT_ICON_DECLARATIONS, float _shadow_thickness, Color_Group* _color)
Engine_Component_Icon(ENGINE_COMPONENT_ICON_DECLARATIONS, char* _name, float _character_height)
Engine_Component_Icon(ENGINE_COMPONENT_ICON_DECLARATIONS, float _shadow_thickness, Color_Group* _color, char* _name, float _character_height)
_color ..... *Color information object*
_shadow_thickness ..... *Thickness of bevel shadow*
_name ..... *Name of icon to be used for the label*
_character_height ..... *Height of the label text in modeling coordinates*
void activate(int _perform_flag)
Activate icon. This function sets the state flag to ACTIVE. It also creates the illusion of lowering the push button by changing the bevel shadow colors.
_perform_flag ..... *Update display? PERFORM or DO_NOT_PERFORM*
void deactivate(int _perform_flag)
Deactivate icon. This function sets the state flag to INACTIVE. It also creates the illusion of raising the push button by changing the bevel shadow colors.
_perform_flag ..... *Update display? PERFORM or DO_NOT_PERFORM*
void ETC_display(void)
This function duplicates the **display_change** function in immediate mode graphics.
void set_name(const char *_name, float _character_height)
This function sets the name, calculates the new dimensions for the button and recreates the icon with the new name as a label.
_name ..... *The new icon name*
_character_height ..... *The character height of the new name label*

Class Name: Flow_Connection_Icon

Class Description: This class contains information and methods used to draw and manipulate the icon for flow connections. A flow connection icon consists of a push button with label and arrows pointing to and/or away from the center of the push button. The arrows are anchored at the coordinates of the connecting upstream and downstream components. The icon indicates the selected state by highlighting the push button and the activated state by lowering the button. During translation of the icon the arrow anchors do not move but the rest of the icon does.

Header file Name: flow_connection_icon.h

Derived from: Icon

Member Classes:

Member Data:

private:

const float** upstream_anchor_x_ptr ..... *Array of pointers to the connected upstream components' x coordinates*

const float** upstream_anchor_y_ptr ..... *Array of pointers to the connected upstream components' y coordinates*

const float** upstream_anchor_width_ptr ..... *Array of pointers to the connected upstream components' widths*

const float** upstream_anchor_height_ptr ..... *Array of pointers to the connected upstream components' heights*

const float** downstream_anchor_x_ptr ..... *Array of pointers to the connected downstream components' x coordinates*

const float** downstream_anchor_y_ptr ..... *Array of pointers to the connected downstream components' y coordinates*

const float** downstream_anchor_width_ptr ..... *Array of pointers to the connected downstream components' widths*

***const float** downstream_anchor_height_ptr*.....*Array of pointers to the connected downstream components' heights*

Member Functions:

FLOW_CONNECTION_ICON_DECLARATIONS is defined as the following list of arguments:

float _x, float _y, int _horizontal_alignment, int _vertical_alignment, float _width, float _height

_x.....*x location of icon's reference point*

_y.....*y location of icon's reference point*

_horizontal_alignment.....*Position of reference point on icon: LEFT, CENTER, or RIGHT*

_vertical_alignment.....*Position of reference point on icon: TOP, CENTER, or BOTTOM*

_width.....*Width of icon in modeling coordinates*

_height.....*Height of icon in modeling coordinates*

private:

void initializer(FLOW_CONNECTION_ICON_DECLARATIONS, float _shadow_thickness, Color_Group* _color)

Initialize icon data.

_shadow_thickness.....*Thickness of bevel shadow*

_color.....*Color information object*

void create_structure(void)

Open a PHIGS structure. Call the functions to create the pieces of the icon. Close the structure.

void create_name(void)

Set text attributes and execute a PHIGS text primitive with the icon name as argument.

void create_lower_right_bevel(void)

Set polygon attributes and execute a PHIGS polygon primitive for the lower and right bevel of the push button slab.

void create_upper_left_bevel(void)

Set polygon attributes and execute a PHIGS polygon primitive for the upper and left bevel of the push button slab.

void create_face(void)

Set polygon attributes and execute a PHIGS polygon primitive for the face of the push button slab.

void create_arrow(float _x1, float _y1, float _x2, float _y2)

Set attributes and execute PHIGS primitives to draw a line with a centered arrowhead from point 1 to point 2 as defined by the arguments. The color of the arrow and line is the same as the text color.

_x1, _y1.....*Coordinates of first point*

_x2, _y2.....*Coordinates of second point*

public:

Flow_Connection_Icon(FLOW_CONNECTION_ICON_DECLARATIONS)

Flow_Connection_Icon(FLOW_CONNECTION_ICON_DECLARATIONS, Color_Group* _color)

Flow_Connection_Icon(FLOW_CONNECTION_ICON_DECLARATIONS, float _shadow_thickness)

Flow_Connection_Icon(FLOW_CONNECTION_ICON_DECLARATIONS, float _shadow_thickness, Color_Group* _color)

Flow_Connection_Icon(FLOW_CONNECTION_ICON_DECLARATIONS, char* _name, float _character_height)

Flow_Connection_Icon(FLOW_CONNECTION_ICON_DECLARATIONS, float _shadow_thickness, Color_Group* _color, char* _name, float _character_height)

_color.....*Color information object*

_shadow_thickness.....*Thickness of bevel shadow*

_name ..... *Name of icon to be used for the label*
_character_height ..... *Height of the label text in modeling coordinates*

void activate(int _perform_flag)

Activate icon. This function sets the state flag to ACTIVE. It also creates the illusion of lowering the push button by changing the bevel shadow colors.

_perform_flag ..... *Update display? PERFORM or DO_NOT_PERFORM*

void deactivate(int _perform_flag)

Deactivate icon. This function sets the state flag to INACTIVE. It also creates the illusion of raising the push button by changing the bevel shadow colors.

_perform_flag ..... *Update display? PERFORM or DO_NOT_PERFORM*

void ETC_display(void)

This function duplicates the **display_change** function in immediate mode graphics.

void set_name(const char *_name, float _character_height)

This function sets the name, calculates the new dimensions for the button and recreates the icon with the new name as a label.

_name ..... *The new icon name*
_character_height ..... *The character height of the new name label*

void add_upstream_arrow(const float* _x_ptr, const float* _y_ptr, const float* _width_ptr, const float* _height_ptr)

This function adds an upstream arrow to the icon by allocating memory for the coordinates and dimensions in their arrays and updating the graphics structure.

_x_ptr, _y_ptr ..... *Pointers to the coordinates of the connecting component's reference point*
_width_ptr, _height_ptr ..... *Pointers to the dimensions of the connecting component*

void add_downstream_arrow(const float* _x_ptr, const float* _y_ptr, const float* _width_ptr, const float* _height_ptr)

This function adds an downstream arrow to the icon by allocating memory for the coordinates and dimensions in their arrays and updating the graphics structure.

_x_ptr, _y_ptr ..... *Pointers to the coordinates of the connecting component's reference point*
_width_ptr, _height_ptr ..... *Pointers to the dimensions of the connecting component*

void remove_arrow(const float* _x_ptr, const float* _y_ptr)

This function removes an arrow from the icon by reclaiming memory for the coordinates and dimensions from their arrays and updating the graphics structure. The arrow is identified by matching the arguments to the anchor point of the arrow

_x_ptr, _y_ptr ..... *Pointers to the coordinates of the arrow's anchor point*

Class Name: Rotational_Connection_Icon

Class Description: This class contains information and methods needed to draw and manipulate the icon for a rotational connection. The icon consists of two parallel lines joined at the coordinates of the icon. The icon does not have a visual response to selection or activation. The icon can only be translated as a function of the position of its anchor points.

Header file Name: rotational_connection_icon.h

Derived from: Icon

Member Classes:

Member Data:

const float** anchor_x_ptr ..... *Array of pointers to the connected components' x coordinates*

const float** anchor_y_ptr ..... *Array of pointers to the connected components' y coordinates*

const float** anchor_width_ptr ..... *Array of pointers to the connected components' widths*

const float** anchor_height_ptr ..... *Array of pointers to the connected components' heights*

Member Functions:

ROTATIONAL_CONNECTION_ICON_DECLARATIONS is defined as the following list of arguments:
float _x, float _y, int _horizontal_alignment, int _vertical_alignment, float _width, float _height

_x ..... *x location of icon's reference point*

_y ..... *y location of icon's reference point*

_horizontal_alignment ..... *Position of reference point on icon: LEFT, CENTER, or RIGHT*

_vertical_alignment ..... *Position of reference point on icon: TOP, CENTER, or BOTTOM*

_width ..... *Width of icon in modeling coordinates*

_height ..... *Height of icon in modeling coordinates*

private:

void initializer(ROTATIONAL_CONNECTION_ICON_DECLARATIONS, float _shadow_thickness, Color_Group* _color)

Initialize icon data.

_shadow_thickness.....*Thickness of bevel shadow*

_color.....*Color information object*

void create_structure(void)

Open a PHIGS structure. Call the functions to create the pieces of the icon. Close the structure.

void create_line(float _x1, float _y1, float _x2, float _y2)

Set attributes and execute PHIGS primitive to draw a line from point 1 to point 2. The line color is the text color.

public:

Rotational_Connection_Icon(ROTATIONAL_CONNECTION_ICON_DECLARATIONS)

Rotational_Connection_Icon(ROTATIONAL_CONNECTION_ICON_DECLARATIONS, Color_Group* _color)

Rotational_Connection_Icon(ROTATIONAL_CONNECTION_ICON_DECLARATIONS, float _shadow_thickness)

Rotational_Connection_Icon(ROTATIONAL_CONNECTION_ICON_DECLARATIONS, float _shadow_thickness, Color_Group* _color)

Rotational_Connection_Icon(ROTATIONAL_CONNECTION_ICON_DECLARATIONS, char* _name, float _character_height)

Rotational_Connection_Icon(ROTATIONAL_CONNECTION_ICON_DECLARATIONS, float _shadow_thickness, Color_Group* _color, char* _name, float _character_height)

_color.....*Color information object*

_shadow_thickness.....*Thickness of bevel shadow*

_name.....*Name of icon to be used for the label*

_character_height.....*Height of the label text in modeling coordinates*

void activate(int _perform_flag)

Activate icon. This function sets the state flag to ACTIVE.

_perform_flag.....*Update display? PERFORM or DO_NOT_PERFORM*

void deactivate(int _perform_flag)

Deactivate icon. This function sets the state flag to INACTIVE.

_perform_flag.....*Update display? PERFORM or DO_NOT_PERFORM*

void select(int _perform_flag)

Select the icon. This means setting the select_state flag to SELECTED.

_perform_flag.....*Update display? PERFORM or DO_NOT_PERFORM*

void unselect(int _perform_flag)

Deselect the icon. This means setting the select_state flag to UNSELECTED.

_perform_flag.....*Update display? PERFORM or DO_NOT_PERFORM*

int add_line(const float* _x_ptr, const float* _y_ptr, const float* _width_ptr, const float* _height_ptr)

Allocate the memory in the pointer arrays for the coordinate and dimension pointers and redraw the icon

int remove_line(const float* _x_ptr, const float* _y_ptr)

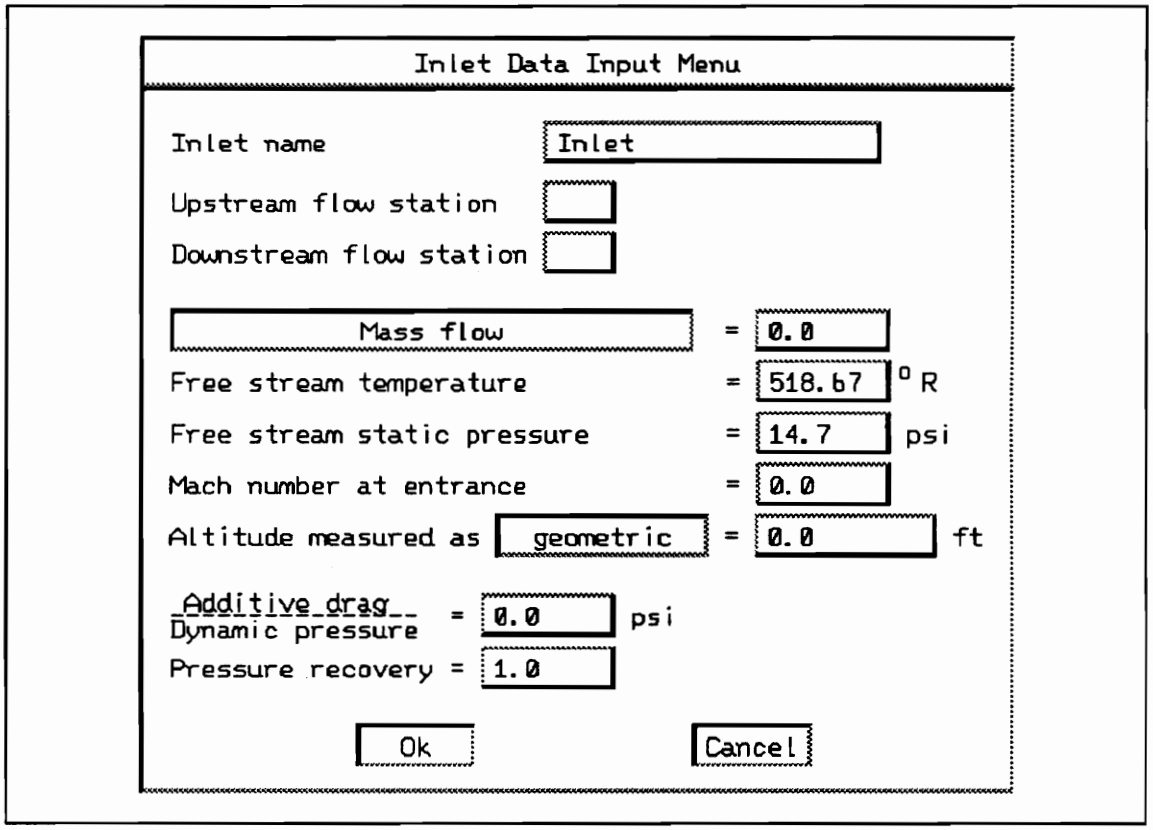
Reclaim the memory in the pointer arrays for the coordinate and dimension pointers and redraw the icon

void ETC_display(void)

This function duplicates the **display_change** function in immediate mode graphics.

Class Name: Inlet

Class Description: The inlet is a specialized engine component. The processing for inlet input data is defined in this class. Further error trapping, as for component/flow station connections, can be added here. The characterizing virtual functions **is_flow_component** and **is_rotational_component** are set in this class as well.



Header file Name: inlet.h

Derived from: Engine_Component

Member Class: none

Member Data:

private:

int mass_flow_index ..... *Index of input field for, depending on the accompanying toggle, mass flow in lbm/s, corrected mass flow at the inlet entrance or corrected mass flow at the inlet exit*

int temperature_index ..... *Index of input field for the free stream temperature*

int pressure_index ..... *Index of input field for the free stream pressure in* lb/in^2

int mach_index ..... *Index of input field for the free stream Mach number*

int altitude_index ..... *Index of input field for, depending on the accompanying toggle, geometric or geopotential altitude in feet*

int drag_index ..... *Index of input field for the ratio of additive drag to dynamic pressure*

int pressure_recovery_index ..... *Index of input field for the inlet pressure recovery*

int mass_flow_type_index ..... *Index of toggle button for the mass flow type: mass flow, inlet entrance corrected mass flow or inlet exit corrected mass flow*

int altitude_type_index ..... *Index of toggle button for altitude type: geometric or geopotential*

Member Functions:

INLET_DECLARATIONS is defined as the following argument list:

Graphics_Info* _graphics_info, const Connection** _connection_list_ptr, float _x, float _y

_graphics_info ..... *Object containing relevant graphics information*

_connection_list_ptr ..... *Pointer to the linked list of connections*

_x, _y ..... *Initial location of the inlet icon*

private:

void initializer(INLET_DECLARATIONS)

The initializer function for the inlet class sets the filename root used for all inlet related file input. It also initializes the name pointer arrays for upstream and downstream connections. It assigns indices into the input menu item arrays and sets the input data defaults stored in the file *.inlet.defaults*.

int mass_flow(char* _string)

int temperature(char* _string)

int pressure(char* _string)

int mach(char* _string)

int altitude(char* _string)

int drag(char* _string)

int pressure_recovery(char* _string)

These functions check the input string for validity as input data. They then set the NEPP data array element corresponding to the input item taking into account toggle button settings and dependencies on other input fields. The functions return REFUSED if the input string contained unacceptable data or was out of bounds.

_string ..... *This is the input string containing the user input for the input field corresponding to the function*

public:

Inlet(void)

Inlet(INLET_DECLARATIONS)

Inlet(INLET_DECLARATIONS, char* _name)

void set_type(void)

This function has to be redefined in each class derived from the base Engine_Component class. It issues an error message that the component type may not be changed.

void set_type(Component_Type _type)

This function resets the component if the argument, _type, is INLET_TYPE. If it is not it issues a warning.

_type ..... *Type of engine component*

void process_push_buttons(Event* _event)

Process push button events. This function toggles through push button options.

_event ..... *Event list possibly containing push button event*

int verify_input(Text_Input** _text_input_item, Push_Button** _button)

Calls the base class **verify_input** function to check name and connection inputs and then all of the input string processing functions from this class. If any one of these error trapping functions returns REFUSED, this function returns refused.

_text_input_item ..... *Array of text input field menu items containing the new user input strings*

_push_button_item ..... *Array of push button menu items containing the new user specified toggle settings*

int is_flow_component(void)

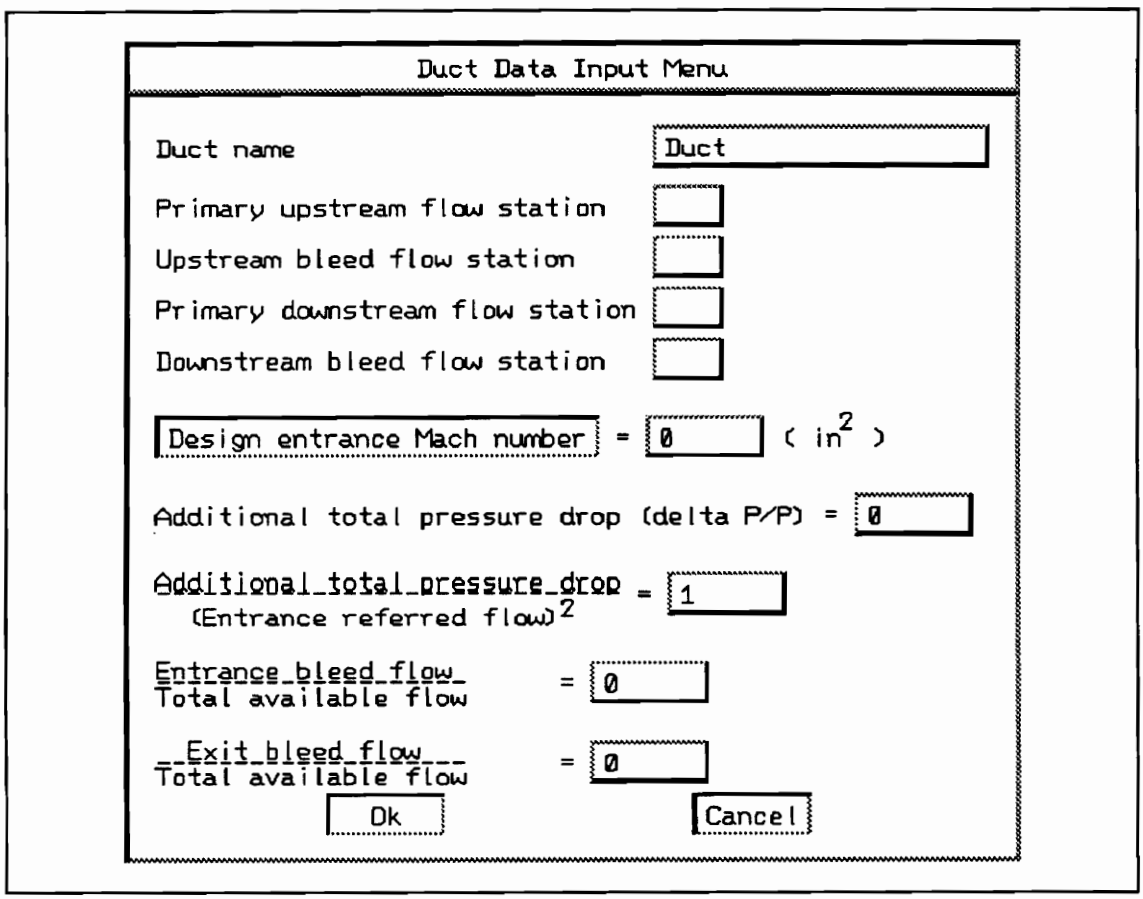
Returns YES.

int is_rotational_component(void)

Returns NO.

Class Name: Duct

Class Description: The duct is a specialized engine component. The processing for duct input data is defined in this class. Further error trapping, as for component/flow station connections, can be added here. The characterizing virtual functions **is_flow_component** and **is_rotational_component** are set in this class as well.



Header file Name: duct.h

Derived from: Engine_Component

Member Class: none

Member Data:

private:

int mach_area_index ..... *Index of input field for, depending on the accompanying toggle, the design Mach number at the duct entrance or the cross-sectional area of the duct*

int pressure_drop_index ..... *Index of input field for the duct pressure drop in addition to the calculated Raleigh total pressure drop:* $\Delta\ p_t/p_t$

int additional_pressure_drop_index ..... *Index of input field for the ratio of the total additional pressure drop to the duct inlet referred flow squared:* $(\Delta p_t/p_t)/(w\sqrt{T}/p)^2$

int entrance_bleed_index ..... *Index of input field for the ratio of entrance bleed flow to total available flow*

int exit_bleed_index ..... *Index of input field for the ratio of exit bleed flow to total available flow*

int mach_area_type_index ..... *Index of toggle button for sizing duct: Design Mach number at the duct entrance or the cross-sectional area of the duct*

Member Functions:

DUCT_DECLARATIONS is defined as the following argument list:

Graphics_Info* _graphics_info, const Connection** _connection_list_ptr, float _x, float _y

_graphics_info ..... *Object containing relevant graphics information*

_connection_list_ptr ..... *Pointer to the linked list of connections*

_x, _y ..... *Initial location of the duct icon*

private:

void initializer(DUCT_DECLARATIONS)

The initializer function for the duct class sets the filename root used for all duct related file input. It also initializes the name pointer arrays for upstream and downstream connections. It assigns indices into the input menu item arrays and sets the input data defaults stored in the file *.duct.defaults*.

int mach(char* _string)

int area(char* _string)

int pressure_drop(char* _string)

int additional_pressure_drop(char* _string)

int entrance_bleed(char* _string)

int exit_bleed(char* _string)

These functions check the input string for validity as input data. They then set the NEPP data array element corresponding to the input item taking into account toggle button settings and dependencies on other input fields. The functions return REFUSED if the input string contained unacceptable data or was out of bounds.

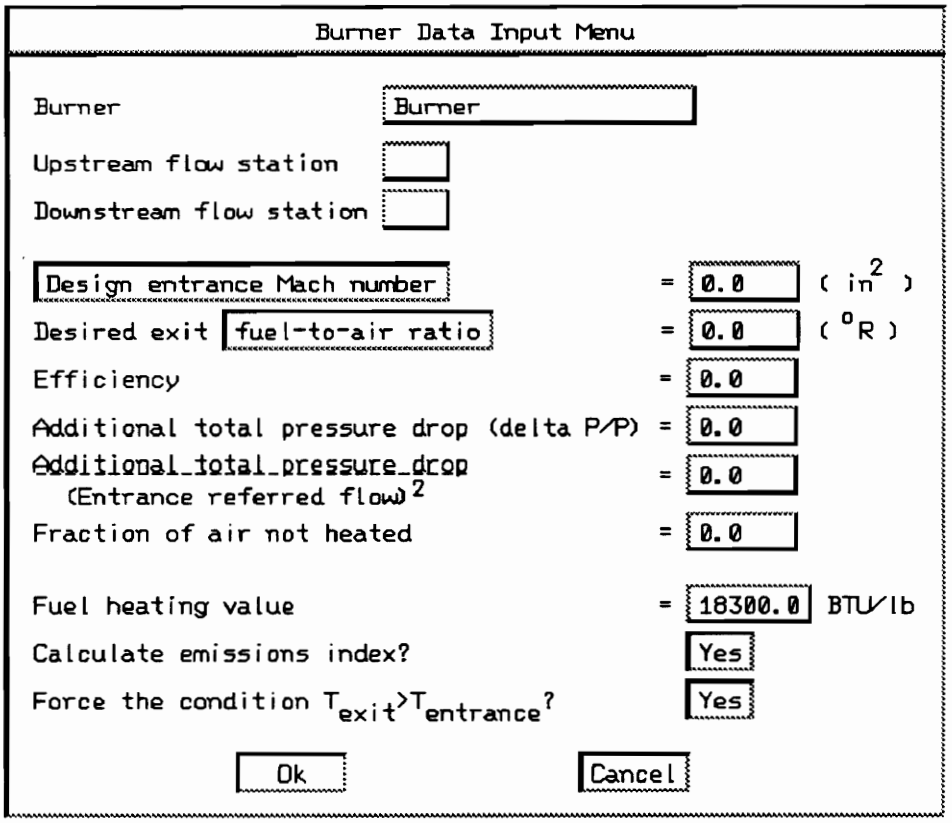
_string ..... *This is the input string containing the user input for the input field corresponding to the function*

public:

Duct(void)
Duct(DUCT_DECLARATIONS)
Duct(DUCT_DECLARATIONS, char* _name)
void set_type(void)
This function has to be redefined in each class derived from the base Engine_Component class. It issues an error message that the component type may not be changed.
void set_type(Component_Type _type)
This function resets the component if the argument, _type, is DUCT_TYPE. If it is not it issues a warning.
_type.....*Type of engine component*
void process_push_buttons(Event* _event)
Process push button events. This function toggles through push button options.
_event.....*Event list possibly containing push button event*
int verify_input(Text_Input** _text_input_item, Push_Button** _button)
Calls the base class **verify_input** function to check name and connection inputs and then all of the input string processing functions from this class. If any one of these error trapping functions returns REFUSED, this function returns refused.
_text_input_item.....*Array of text input field menu items containing the new user input strings*
_push_button_item.....*Array of push button menu items containing the new user specified toggle settings*
int is_flow_component(void)
Returns YES.
int is_rotational_component(void)
Returns NO.

Class Name: Burner

Class Description: The burner is a specialized engine component. The processing for burner input data is defined in this class. Further error trapping, as for component/flow station connections, can be added here. The characterizing virtual functions **is_flow_component** and **is_rotational_component** are set in this class as well.



Header file Name: burner.h

Derived from: Engine_Component

Member Class: none

Member Data:

private:

int mach_area_index ..... *Index of input field for, depending on the accompanying toggle, the design Mach number at the burner entrance or the cross-sectional area of the burner*

int temp_FAratio_index ..... *Index of input field for, depending on the accompanying toggle, the burner temperature or the fuel-to-air ratio*

int efficiency_index ..... *Index of input field for the burner efficiency*

int pressure_drop_index ..... *Index of input field for the burner pressure drop in addition to the calculated Raleigh total pressure drop:* $\Delta\ p_t/p_t$

int pressure_drop_ratio_index ..... *Index of input field for the ratio of the total additional pressure drop to the burner inlet referred flow squared:* $(\Delta p_t/p_t)/(w\sqrt{T}/p)^2$

int unheated_air_index ..... *Index of input field for the fraction of air flow not heated but bypassed and then mixed in with the heated flow at the burner exit*

int heating_value_index ..... *Index of input field for the heating value of the fuel in Btu/lb*

int mach_area_type_index ..... *Index of toggle button for sizing burner: Design Mach number at the burner entrance or the cross-sectional area of the burner*

int temp_FAratio_type_index ..... *Index of toggle button for specifying throttle: temperature or fuel-to-air ratio*

int emissions_index ..... *Index of input field specifying whether or not to calculate the emissions index*

int force_temp_condition_index ..... *Index of input field specifying whether or not to force the exit temperature to exceed the entrance temperature*

Member Functions:

BURNER_DECLARATIONS is defined as the following argument list:

Graphics_Info* _graphics_info, const Connection** _connection_list_ptr, float _x, float _y

_graphics_info ..... *Object containing relevant graphics information*

_connection_list_ptr ..... *Pointer to the linked list of connections*

_x, _y ..... *Initial location of the burner icon*

private:

void initializer(BURNER_DECLARATIONS)

The initializer function for the burner class sets the filename root used for all burner related file input. It also initializes the name pointer arrays for upstream and downstream connections. It assigns indices into the input menu item arrays and sets the input data defaults stored in the file *.burner.defaults*.

int mach(char* _string)

int area(char* _string)
int temperature(char* _string)
int fuel_air(char* _string)
int efficiency(char* _string)
int pressure_drop(char* _string)
int pressure_drop_ratio(char* _string)
int unheated_air(char* _string)
int heating_value(char* _string)
These functions check the input string for validity as input data. They then set the NEPP data array element corresponding to the input item taking into account toggle button settings and dependencies on other input fields. The functions return REFUSED if the input string contained unacceptable data or was out of bounds.

_string ..... *This is the input string containing the user input for the input field corresponding to the function*

public:
Burner(void)
Burner(BURNER_DECLARATIONS)
Burner(BURNER_DECLARATIONS, char* _name)
void set_type(void)
This function has to be redefined in each class derived from the base Engine_Component class. It issues an error message that the component type may not be changed.
void set_type(Component_Type _type)
This function resets the component if the argument, _type, is BURNER_TYPE. If it is not it issues a warning.

_type ..... *Type of engine component*

void process_push_buttons(Event* _event)
Process push button events. This function toggles through push button options.

_event ..... *Event list possibly containing push button event*

int verify_input(Text_Input** _text_input_item, Push_Button** _button)
Calls the base class **verify_input** function to check name and connection inputs and then all of the input string processing functions from this class. If any one of these error trapping functions returns REFUSED, this function returns refused.

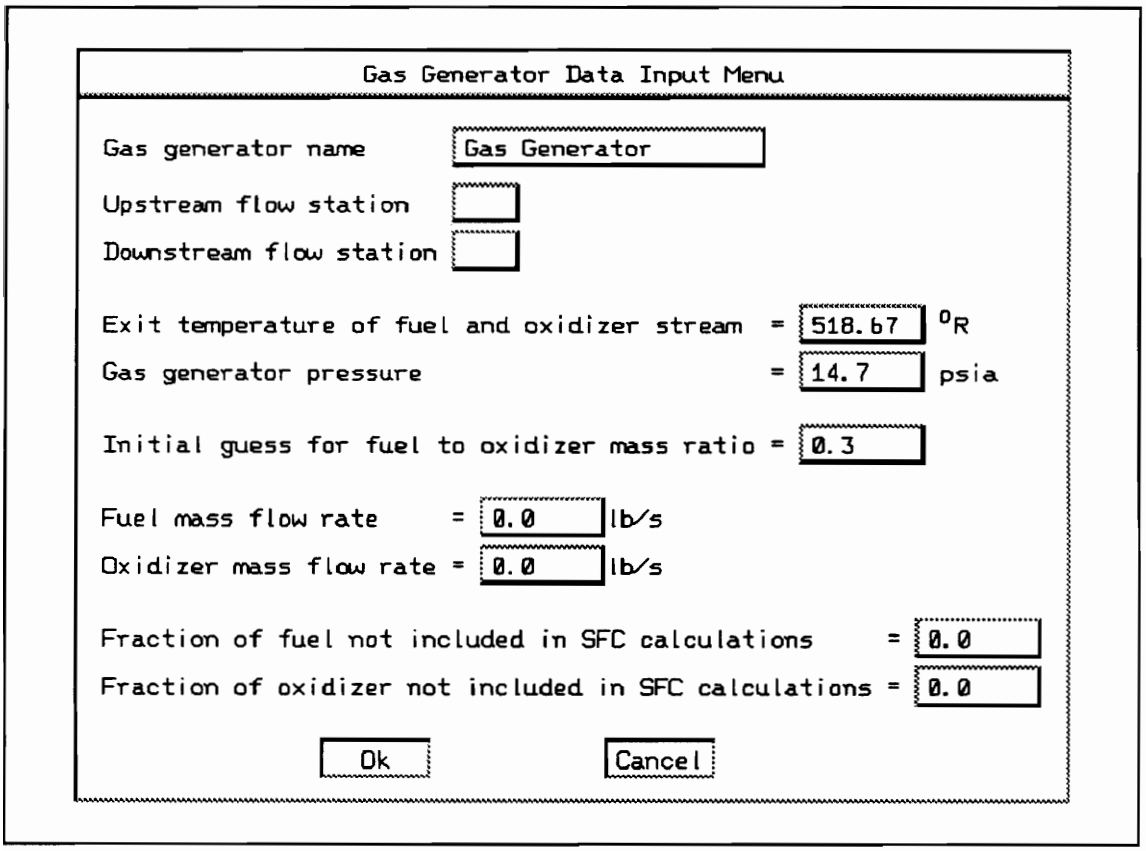
_text_input_item ..... *Array of text input field menu items containing the new user input strings*

_push_button_item ..... *Array of push button menu items containing the new user specified toggle settings*

int is_flow_component(void)
Returns YES.
int is_rotational_component(void)
Returns NO.

Class Name: Gas_Generator

Class Description: The gas generator is a specialized engine component. The processing for gas generator input data is defined in this class. Further error trapping, as for component/flow station connections, can be added here. The characterizing virtual functions **is_flow_component** and **is_rotational_component** are set in this class as well.



Header file Name: gas_generator.h

Derived from: Engine_Component

Member Class: none

Member Data:

private:

int temperature_index ..... *Index of input field for the gas generator exit temperature in °R*

int pressure_index ..... *Index of input field for the gas generator pressure in psia*

int mass_ratio_index ..... *Index of input field for the gas generator fuel-to-oxidant mass ratio*

int fuel_flow_index ..... *Index of input field for the gas generator fuel mass flow in lb/s*

int oxygen_flow_index ..... *Index of input field for the gas generator oxidant mass flow in lb/s*

int fuel_fraction_index ..... *Index of input field for the fraction of fuel not included in the engine fuel consumption calculations*

int oxygen_fraction_index ..... *Index of input field for the fraction of oxidant not included in the engine fuel consumption calculations*

Member Functions:

GAS_GENERATOR_DECLARATIONS is defined as the following argument list:

Graphics_Info* _graphics_info, const Connection** _connection_list_ptr, float _x, float _y

_graphics_info ..... *Object containing relevant graphics information*

_connection_list_ptr ..... *Pointer to the linked list of connections*

_x, _y ..... *Initial location of the gas generator icon*

private:

void initializer(GAS_GENERATOR_DECLARATIONS)

The initializer function for the gas generator class sets the filename root used for all gas generator related file input. It also initializes the name pointer arrays for upstream and downstream connections. It assigns indices into the input menu item arrays and sets the input data defaults stored in the file *.gas_generator.defaults*.

int temperature(char* _string)
int pressure(char* _string)
int mass_flow(char* _string)
int fuel_flow(char* _string)
oxygen_flow(char* _string)
fuel_fraction(char* _string)
oxygen_fraction(char* _string)

These functions check the input string for validity as input data. They then set the NEPP data array element corresponding to the input item taking into account toggle button settings and dependencies on other input fields. The functions return REFUSED if the input string contained unacceptable data or was out of bounds.

_string ..... *This is the input string containing the user input for the input field corresponding to the function*

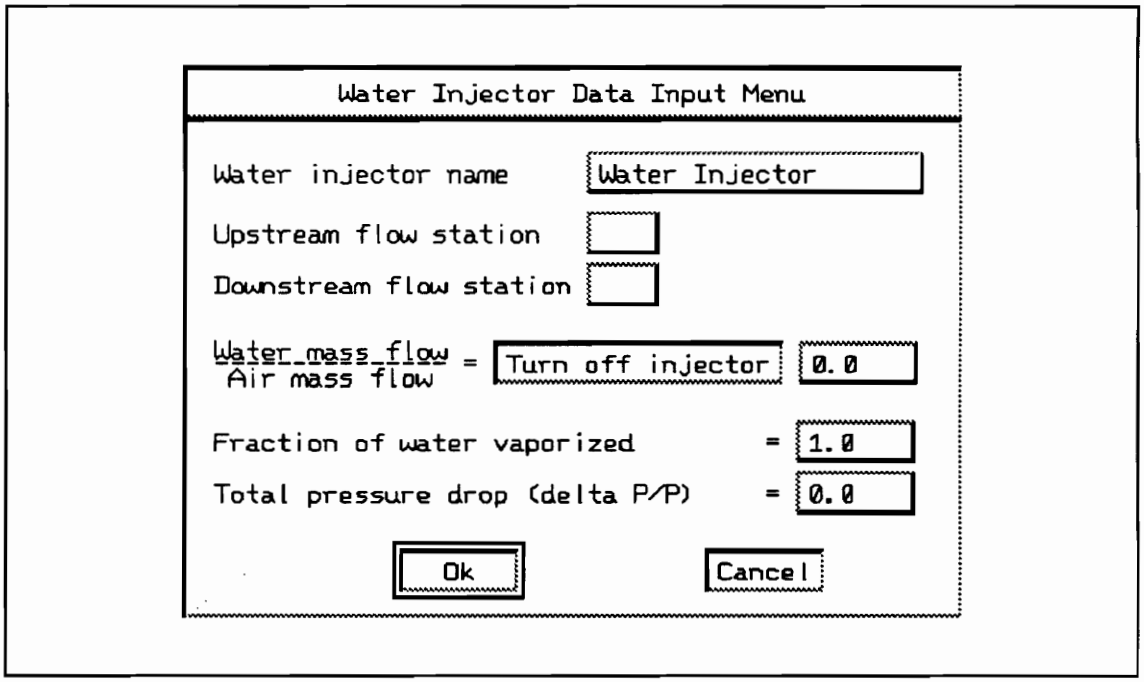
public:

Gas_Generator(void)

Gas_Generator(GAS_GENERATOR_DECLARATIONS)
Gas_Generator(GAS_GENERATOR_DECLARATIONS, char* _name)
void set_type(void)
This function has to be redefined in each class derived from the base Engine_Component class. It issues an error message that the component type may not be changed.
void set_type(Component_Type _type)
This function resets the component if the argument, _type, is GAS_GENERATOR_TYPE. If it is not it issues a warning.
_type.....*Type of engine component*
int verify_input(Text_Input** _text_input_item, Push_Button** _button)
Calls the base class **verify_input** function to check name and connection inputs and then all of the input string processing functions from this class. If any one of these error trapping functions returns REFUSED, this function returns refused.
_text_input_item.....*Array of text input field menu items containing the new user input strings*
_push_button_item.....*Array of push button menu items containing the new user specified toggle settings*
int is_flow_component(void)
Returns YES.
int is_rotational_component(void)
Returns NO.

Class Name: Water_Injector

Class Description: The water injector is a specialized engine component. The processing for water injector input data is defined in this class. Further error trapping, as for component/flow station connections, can be added here. The characterizing virtual functions **is_flow_component** and **is_rotational_component** are set in this class as well.



Header file Name: water_injector.h

Derived from: Engine_Component

Member Class: none

Member Data:

private:

int mass_flow_ratio_index ..... *Index of input field for water-to-air mass flow ratio*

int vaporized_water_index ..... *Index of input field for fraction of water vaporized*

int pressure_drop_index ..... *Index of input field for the water injector total pressure drop:* $\Delta\ p_t/p_t$

int saturate_index ..... *Index of toggle button specifying whether or not to saturate the mixture*

Member Functions:

WATER_INJECTOR_DECLARATIONS is defined as the following argument list:

Graphics_Info* _graphics_info, const Connection** _connection_list_ptr, float _x, float _y

_graphics_info ..... *Object containing relevant graphics information*

_connection_list_ptr ..... *Pointer to the linked list of connections*

_x, _y ..... *Initial location of the water injector icon*

private:

void initializer(WATER_INJECTOR_DECLARATIONS)

The initializer function for the water injector class sets the filename root used for all water injector related file input. It also initializes the name pointer arrays for upstream and downstream connections. It assigns indices into the input menu item arrays and sets the input data defaults stored in the file *.water_injector.defaults*.

int mass_flow_ratio(char* _string)

int vaporized_water(char* _string)

int pressure_drop(char* _string)

These functions check the input string for validity as input data. They then set the NEPP data array element corresponding to the input item taking into account toggle button settings and dependencies on other input fields. The functions return REFUSED if the input string contained unacceptable data or was out of bounds.

_string ..... *This is the input string containing the user input for the input field corresponding to the function*

public:

Water_Injector(void)

Water_Injector(WATER_INJECTOR_DECLARATIONS)

Water_Injector(WATER_INJECTOR_DECLARATIONS, char* _name)

void set_type(void)

This function has to be redefined in each class derived from the base Engine_Component class. It issues an error message that the component type may not be changed.

void set_type(Component_Type _type)

This function resets the component if the argument, _type, is WATER_INJECTOR_TYPE. If it is not it issues a warning.

_type ..... *Type of engine component*

void process_push_buttons(Event* _event)

Process push button events. This function toggles through push button options.

_event ..... *Event list possibly containing push button event*

int verify_input(Text_Input** _text_input_item, Push_Button** _button)

Calls the base class **verify_input** function to check name and connection inputs and then all of the input string processing functions from this class. If any one of these error trapping functions returns REFUSED, this function returns refused.

_text_input_item ..... *Array of text input field menu items containing the new user input strings*

_push_button_item ..... *Array of push button menu items containing the new user specified toggle settings*

int is_flow_component(void)
Returns YES.

int is_rotational_component(void)
Returns NO.

Class Name: Fan

Class Description: The fan is a specialized engine component. The processing for fan input data is defined in this class. Further error trapping, as for component/flow station connections, can be added here. The characterizing virtual functions **is_flow_component** and **is_rotational_component** are set in this class as well.

Header file Name: fan.h

Derived from: Engine_Component

Member Class: none

Member Data:

private:

int pressure_ratio_index ..... *Index of input field for total pressure ratio across fan*

int efficiency_index ..... *Index of input field for the fan efficiency*

int weight_flow_index ..... *Index of input field for the corrected weight flow*

int bleed_flow_ratio_index ..... *Index of input field for the ratio of fan bleed flow to total flow into the fan*

Member Functions:

FAN_DECLARATIONS is defined as the following argument list:

Graphics_Info* _graphics_info, const Connection** _connection_list_ptr, float _x, float _y

_graphics_info ..... *Object containing relevant graphics information*

_connection_list_ptr ..... *Pointer to the linked list of connections*

_x, _y ..... *Initial location of the fan icon*

private:

void initializer(FAN_DECLARATIONS)

The initializer function for the fan class sets the filename root used for all fan related file input. It also initializes the name pointer arrays for upstream and downstream connections. It assigns indices into the input menu item arrays and sets the input data defaults stored in the file *.fan.defaults*.

int pressure_ratio(char* _string)

int efficiency(char* _string)

int weight_flow(char* _string)

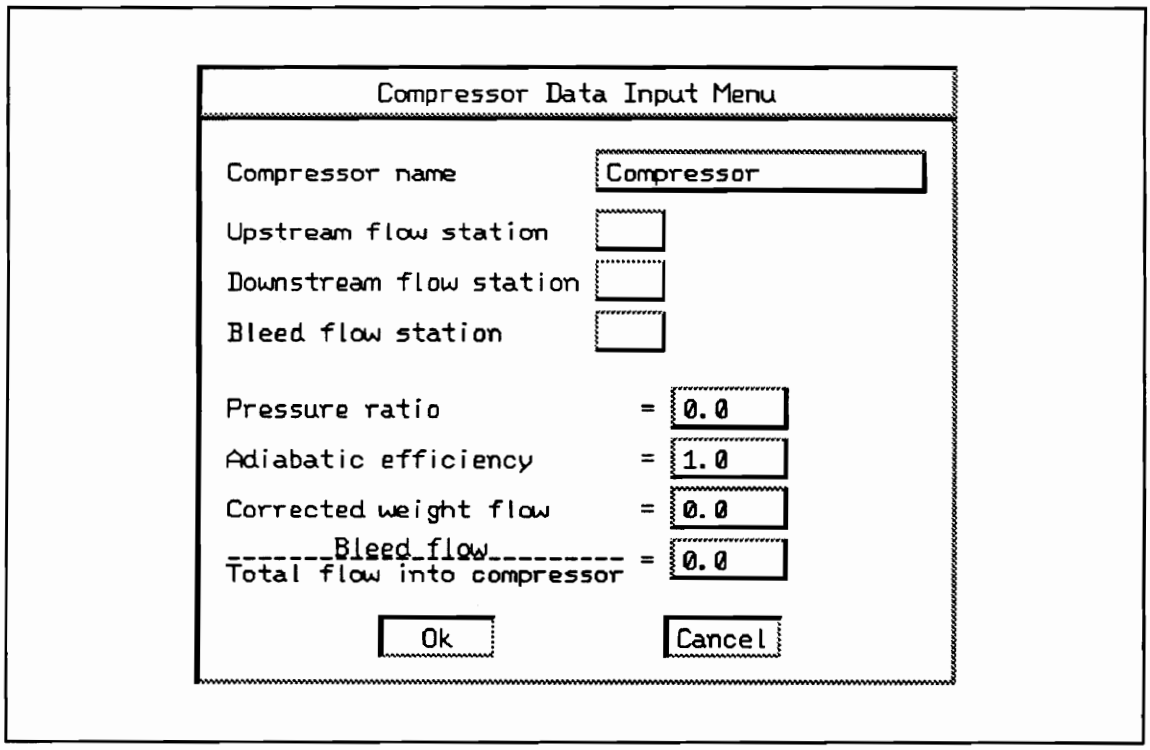
int bleed_flow_ratio(char* _string)
These functions check the input string for validity as input data. They then set the NEPP data array element corresponding to the input item taking into account toggle button settings and dependencies on other input fields. The functions return REFUSED if the input string contained unacceptable data or was out of bounds.
_string ..... *This is the input string containing the user input for the input field corresponding to the function*

public:
Fan(void)
Fan(FAN_DECLARATIONS)
Fan(FAN_DECLARATIONS, char* _name)
void set_type(void)
This function has to be redefined in each class derived from the base Engine_Component class. It issues an error message that the component type may not be changed.
void set_type(Component_Type _type)
This function resets the component if the argument, _type, is FAN_TYPE. If it is not it issues a warning.
_type ..... *Type of engine component*
int verify_input(Text_Input** _text_input_item, Push_Button** _button)
Calls the base class **verify_input** function to check name and connection inputs and then all of the input string processing functions from this class. If any one of these error trapping functions returns REFUSED, this function returns refused.
_text_input_item ..... *Array of text input field menu items containing the new user input strings*
_push_button_item ..... *Array of push button menu items containing the new user specified toggle settings*

int is_flow_component(void)
Returns YES.
int is_rotational_component(void)
Returns YES.

Class Name: Compressor

Class Description: The compressor is a specialized engine component. The processing for compressor input data is defined in this class. Further error trapping, as for component/flow station connections, can be added here. The characterizing virtual functions **is_flow_component** and **is_rotational_component** are set in this class as well.



Header file Name: compressor.h

Derived from: Engine_Component

Member Class: none

Member Data:

private:

int pressure_ratio_index ..... *Index of input field for total pressure ratio across compressor*

int efficiency_index ..... *Index of input field for the compressor efficiency*

int weight_flow_index ..... *Index of input field for the corrected weight flow*

int bleed_flow_ratio_index ..... *Index of input field for the ratio of compressor bleed flow to total flow into the compressor*

Member Functions:

COMPRESSOR_DECLARATIONS is defined as the following argument list:

Graphics_Info* _graphics_info, const Connection** _connection_list_ptr, float _x, float _y

_graphics_info ..... *Object containing relevant graphics information*

_connection_list_ptr ..... *Pointer to the linked list of connections*

_x, _y ..... *Initial location of the compressor icon*

private:

void initializer(COMPRESSOR_DECLARATIONS)

The initializer function for the compressor class sets the filename root used for all compressor related file input. It also initializes the name pointer arrays for upstream and downstream connections. It assigns indices into the input menu item arrays and sets the input data defaults stored in the file *.compressor.defaults*.

int pressure_ratio(char* _string)

int efficiency(char* _string)

int weight_flow(char* _string)

int bleed_flow_ratio(char* _string)

These functions check the input string for validity as input data. They then set the NEPP data array element corresponding to the input item taking into account toggle button settings and dependencies on other input fields. The functions return REFUSED if the input string contained unacceptable data or was out of bounds.

_string ..... *This is the input string containing the user input for the input field corresponding to the function*

public:

Compressor(void)

Compressor(COMPRESSOR_DECLARATIONS)

Compressor(COMPRESSOR_DECLARATIONS, char* _name)

void set_type(void)

This function has to be redefined in each class derived from the base Engine_Component class. It issues an error message that the component type may not be changed.

void set_type(Component_Type _type)

This function resets the component if the argument, _type, is COMPRESSOR_TYPE. If it is not it issues a warning.

_type ..... *Type of engine component*

int verify_input(Text_Input** _text_input_item, Push_Button** _button)

Calls the base class **verify_input** function to check name and connection inputs and then all of the input string processing functions from this class. If any one of these error trapping functions returns REFUSED, this function returns refused.

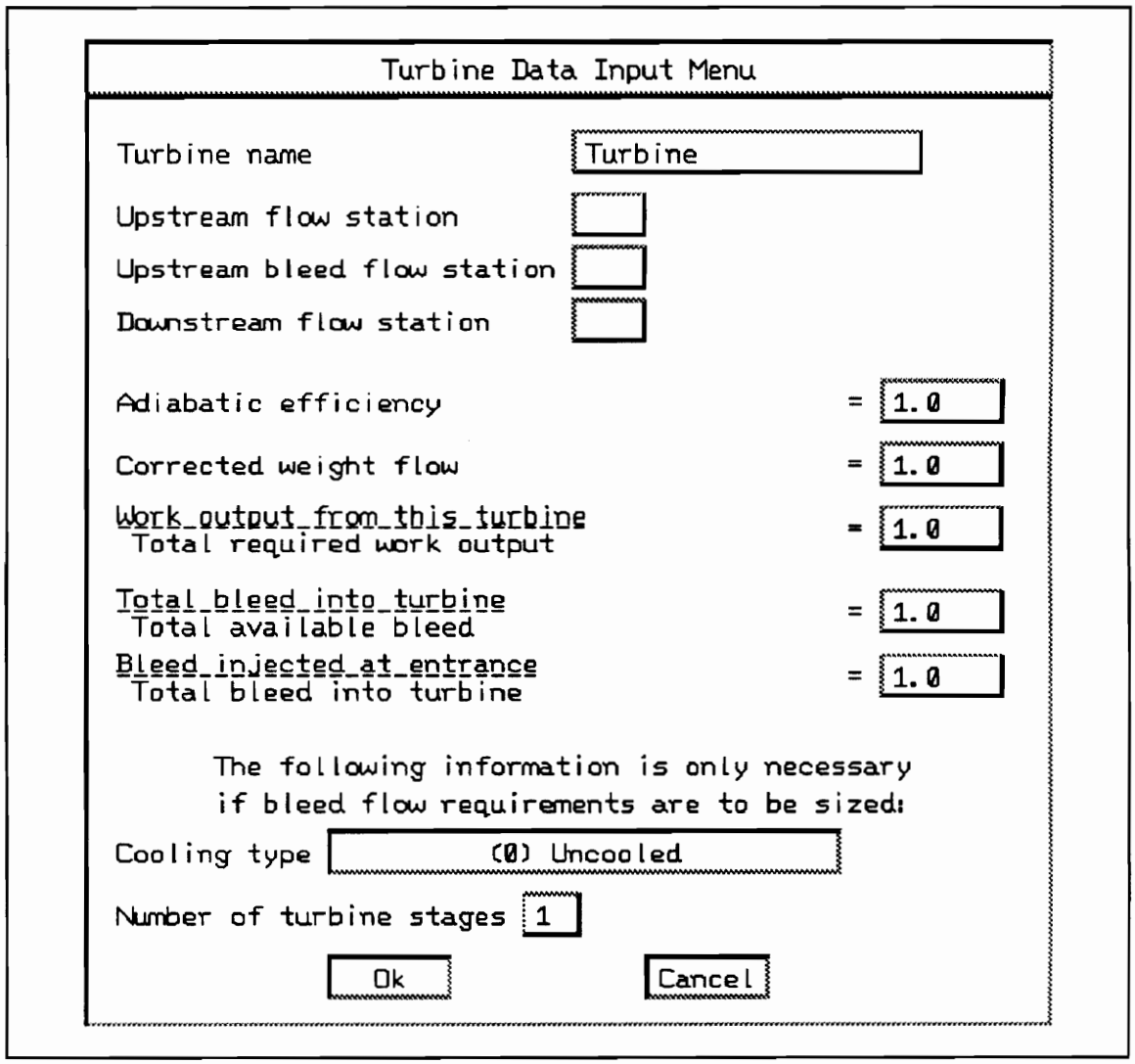
_text_input_item ..... *Array of text input field menu items containing the new user input strings*

_push_button_item.....*Array of push button menu items containing the new user specified toggle settings*

int is_flow_component(void)
Returns YES.
int is_rotational_component(void)
Returns YES.

Class Name: Turbine

Class Description: The turbine is a specialized engine component. The processing for turbine input data is defined in this class. Further error trapping, as for component/flow station connections, can be added here. The characterizing virtual functions **is_flow_component** and **is_rotational_component** are set in this class as well.



Header file Name: turbine.h

Derived from: Engine_Component

Member Class: none

Member Data:

private:

int efficiency_index ..... *Index of input field for the turbine adiabatic efficiency*

int weight_flow_index ..... *Index of input field for the corrected weight flow*

int work_index ..... *Index of input field for the fraction of total work required this turbine must produce*

int total_bleed_ratio_index ..... *Index of input field for the ratio of total bleed into the turbine to total bleed available*

int entrance_bleed_ratio_index ..... *Index of input field for the ratio of bleed injected at the turbine entrance to total bleed injected*

int number_of_stages_index ..... *Index of input field for the number of turbine stages*

int cooling_type_index ..... *Index of toggle button for specifying cooling methods: (0) Uncooled, (1) Convection, (2) Convection with coating, (3) Advanced Convection, (4) Weak film with convection, (5) Medium film with convection, (6) Strong film with convection, (7) Transpiration with convection, (8) Full cover film, (9) Transpiration*

Member Functions:

TURBINE_DECLARATIONS is defined as the following argument list:

Graphics_Info* _graphics_info, const Connection** _connection_list_ptr, float _x, float _y

_graphics_info ..... *Object containing relevant graphics information*

_connection_list_ptr ..... *Pointer to the linked list of connections*

_x, _y ..... *Initial location of the turbine icon*

private:

void initializer(TURBINE_DECLARATIONS)

The initializer function for the turbine class sets the filename root used for all turbine related file input. It also initializes the name pointer arrays for upstream and downstream connections. It assigns indices into the input menu item arrays and sets the input data defaults stored in the file *.turbine.defaults*.

int efficiency(char* _string)

int weight_flow(char* _string)

int work(char* _string)

int total_bleed_ratio(char* _string)

int entrance_bleed_ratio(char* _string)

int number_of_stages(char* _string)

These functions check the input string for validity as input data. They then set the NEPP data array element corresponding to the input item taking into account toggle button settings and dependencies on other input fields. The functions return REFUSED if the input string contained unacceptable data or was out of bounds.

_string ..... *This is the input string containing the user input for the input field corresponding to the function*

public:

Turbine(void)

Turbine(TURBINE_DECLARATIONS)

Turbine(TURBINE_DECLARATIONS, char* _name)

void set_type(void)

This function has to be redefined in each class derived from the base Engine_Component class. It issues an error message that the component type may not be changed.

void set_type(Component_Type _type)

This function resets the component if the argument, _type, is TURBINE_TYPE. If it is not it issues a warning.

_type ..... *Type of engine component*

void process_push_buttons(Event* _event)

Process push button events. This function toggles through push button options.

_event ..... *Event list possibly containing push button event*

int verify_input(Text_Input** _text_input_item, Push_Button** _button)

Calls the base class **verify_input** function to check name and connection inputs and then all of the input string processing functions from this class. If any one of these error trapping functions returns REFUSED, this function returns refused.

_text_input_item ..... *Array of text input field menu items containing the new user input strings*

_push_button_item ..... *Array of push button menu items containing the new user specified toggle settings*

int is_flow_component(void)

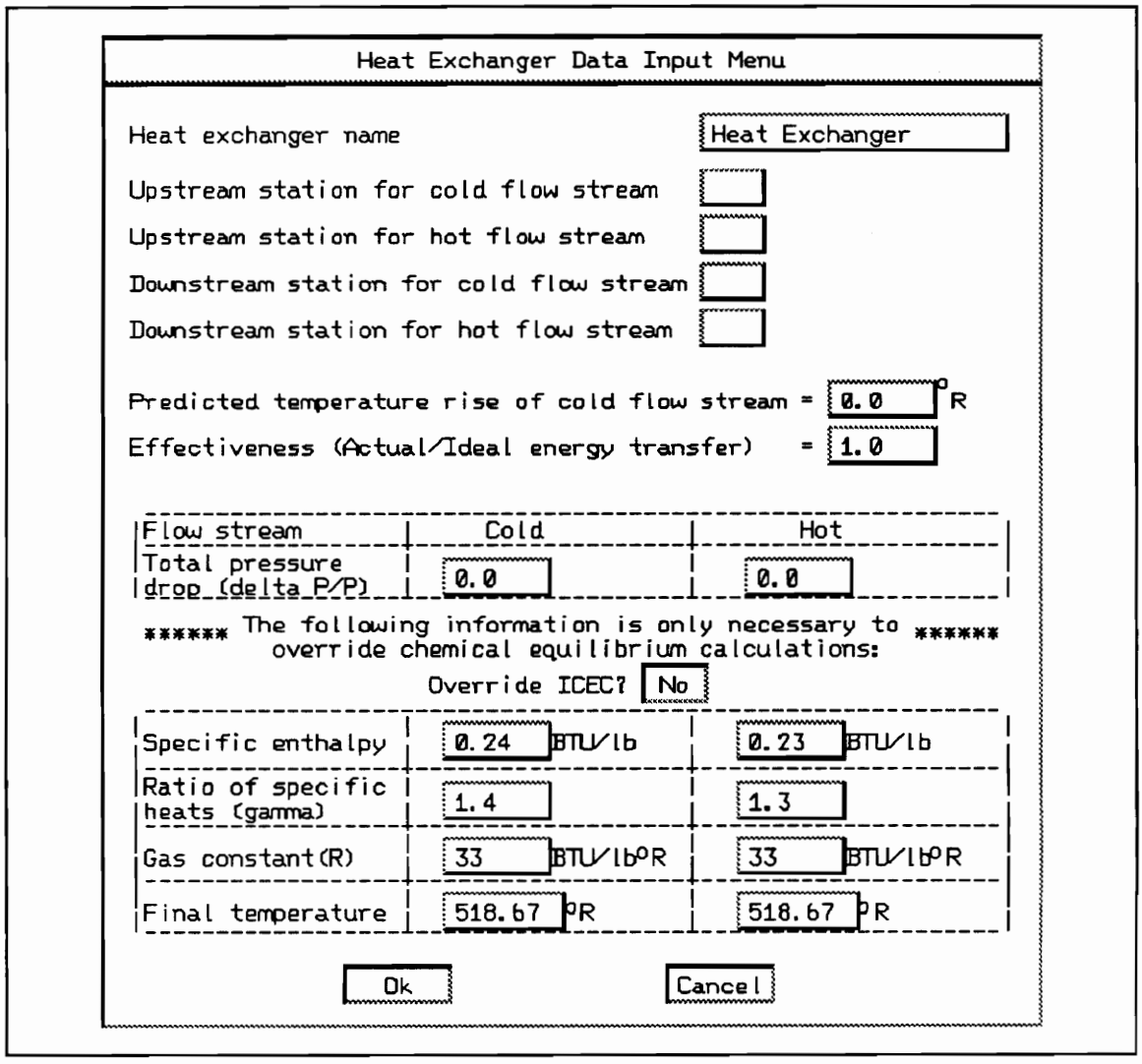
Returns YES.

int is_rotational_component(void)

Returns YES.

Class Name: Heat_Exchanger

Class Description: The heat exchanger is a specialized engine component. The processing for heat exchanger input data is defined in this class. Further error trapping, as for component/flow station connections, can be added here. The characterizing virtual functions **is_flow_component** and **is_rotational_component** are set in this class as well.



Header file Name: heat_exchanger.h

Derived from: Engine_Component

Member Class: none

Member Data:

private:

int temperature_rise_index ..... *Index of input field for a guess value for the temperature rise of the main stream in degrees Rankine*

int effectiveness_index ..... *Index of input field for the heat exchanger effectiveness (ratio of actual thermal energy transfer to ideal thermal energy transfer)*

int heated_pressure_drop_index ..... *Index of input field for the total pressure drop of the heated stream:* $(\Delta p_t/p_t)_h$

int heated_enthalpy_index ..... *Index of input field for the specific enthalpy of the heated flow stream*

int heated_gamma_index ..... *Index of input field for the ratio of specific heats of the heated flow stream*

int heated_R_index ..... *Index of input field for the gas constant of the heated flow stream*

int heated_T_index ..... *Index of input field for the temperature of the heated flow stream in Rankine*

int cooled_pressure_drop_index ..... *Index of input field for the total pressure drop of the cooled stream:* $(\Delta p_t/p_t)_h$

int cooled_enthalpy_index ..... *Index of input field for the specific enthalpy of the cooled flow stream*

int cooled_gamma_index ..... *Index of input field for the ratio of specific heats of the cooled flow stream*

int cooled_R_index ..... *Index of input field for the gas constant of the cooled flow stream*

int cooled_T_index ..... *Index of input field for the temperature of the cooled flow stream in Rankine*

int override_ICEC_index ..... *Index of toggle button for whether to override the chemical equilibrium code values for enthalpy, gamma, R, and temperature for the heated and cooled streams with the given values: YES or NO*

Member Functions:

HEAT_EXCHANGER_DECLARATIONS is defined as the following argument list:

Graphics_Info* _graphics_info, const Connection** _connection_list_ptr, float _x, float _y

_graphics_info ..... *Object containing relevant graphics information*

_connection_list_ptr ..... *Pointer to the linked list of connections*
_x, _y ..... *Initial location of the heat exchanger icon*

private:

void initializer(HEAT_EXCHANGER_DECLARATIONS)

The initializer function for the heat exchanger class sets the filename root used for all heat exchanger related file input. It also initializes the name pointer arrays for upstream and downstream connections. It assigns indices into the input menu item arrays and sets the input data defaults stored in the file *.heat_exchanger.defaults*.

int temperature_rise(char* _string)
int effectiveness(char* _string)
int heated_pressure_drop(char* _string)
int heated_enthalpy(char* _string)
int heated_gamma(char* _string)
int heated_R(char* _string)
int heated_T(char* _string)
int cooled_pressure_drop(char* _string)
int cooled_enthalpy(char* _string)
int cooled_gamma(char* _string)
int cooled_R(char* _string)
int cooled_T(char* _string)

These functions check the input string for validity as input data. They then set the NEPP data array element corresponding to the input item taking into account toggle button settings and dependencies on other input fields. The functions return REFUSED if the input string contained unacceptable data or was out of bounds.

_string ..... *This is the input string containing the user input for the input field corresponding to the function*

public:

Heat_Exchanger(void)
Heat_Exchanger(HEAT_EXCHANGER_DECLARATIONS)
Heat_Exchanger(HEAT_EXCHANGER_DECLARATIONS, char* _name)
void set_type(void)

This function has to be redefined in each class derived from the base Engine_Component class. It issues an error message that the component type may not be changed.

void set_type(Component_Type _type)

This function resets the component if the argument, _type, is HEAT_EXCHANGER_TYPE. If it is not it issues a warning.

_type ..... *Type of engine component*

void process_push_buttons(Event* _event)

Process push button events. This function toggles through push button options.

_event ..... *Event list possibly containing push button event*

int verify_input(Text_Input** _text_input_item, Push_Button** _button)

Calls the base class **verify_input** function to check name and connection inputs and then all of the input string processing functions from this class. If any one of these error trapping functions returns REFUSED, this function returns refused.

_text_input_item ..... *Array of text input field menu items containing the new user input strings*

_push_button_item ..... *Array of push button menu items containing the new user specified toggle settings*

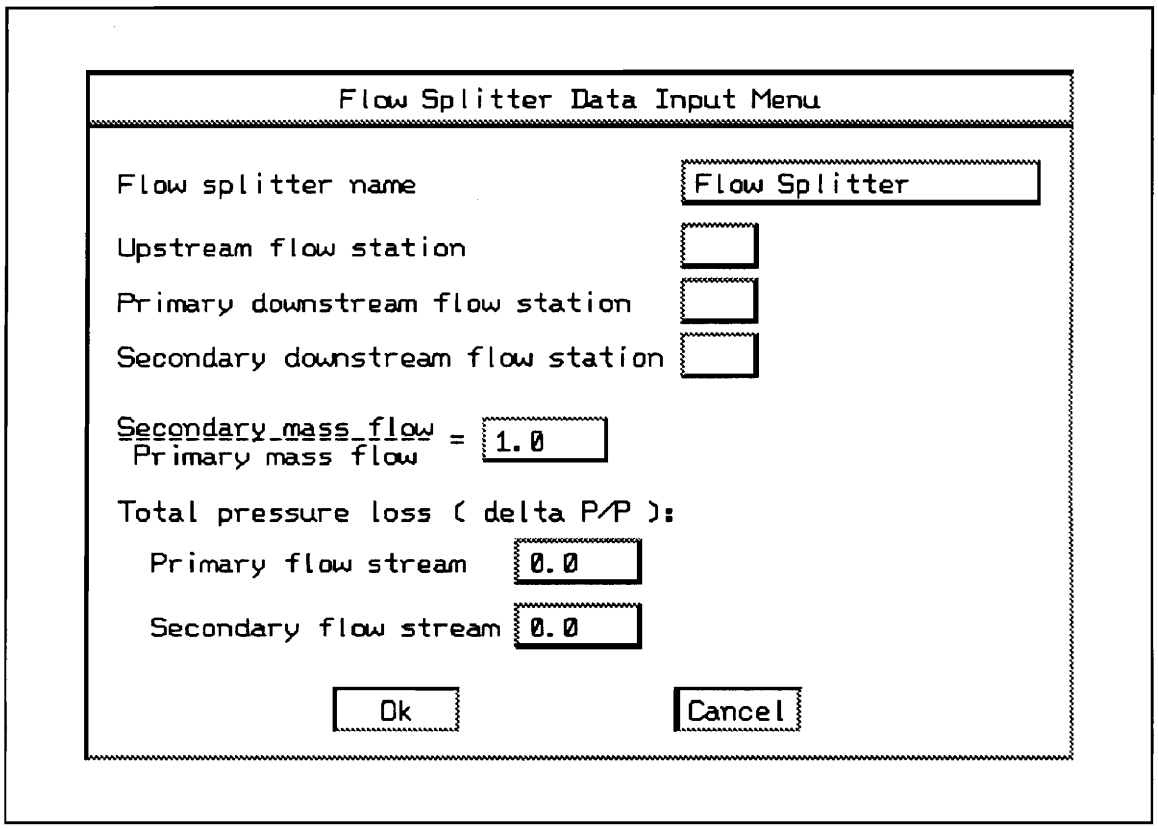
int is_flow_component(void)

Returns YES.

int is_rotational_component(void)
Returns NO.

Class Name: Flow_Splitter

Class Description: The flow splitter is a specialized engine component. The processing for flow splitter input data is defined in this class. Further error trapping, as for component/flow station connections, can be added here. The characterizing virtual functions **is_flow_component** and **is_rotational_component** are set in this class as well.



Header file Name: flow_splitter.h

Derived from: Engine_Component

Member Class: none

Member Data:

private:

int bypass_ratio_index ..... *Index of input field for the ratio of secondary downstream mass flow to main downstream mass flow*

int primary_pressure_drop_index ..... *Index of input field for the main flow stream pressure drop*

int secondary_pressure_drop_index ..... *Index of input field for the secondary flow stream pressure drop*

Member Functions:

FLOW_SPLITTER_DECLARATIONS is defined as the following argument list:

Graphics_Info* _graphics_info, const Connection** _connection_list_ptr, float _x, float _y

_graphics_info ..... *Object containing relevant graphics information*

_connection_list_ptr ..... *Pointer to the linked list of connections*

_x, _y ..... *Initial location of the flow splitter icon*

private:

void initializer(FLOW_SPLITTER_DECLARATIONS)

The initializer function for the flow splitter class sets the filename root used for all flow splitter related file input. It also initializes the name pointer arrays for upstream and downstream connections. It assigns indices into the input menu item arrays and sets the input data defaults stored in the file *.flow_splitter.defaults*.

int bypass_ratio(char* _string)

int primary_pressure_drop(char* _string)

int secondary_pressure_drop(char* _string)

These functions check the input string for validity as input data. They then set the NEPP data array element corresponding to the input item taking into account toggle button settings and dependencies on other input fields. The functions return REFUSED if the input string contained unacceptable data or was out of bounds.

_string ..... *This is the input string containing the user input for the input field corresponding to the function*

public:

Flow_Splitter(void)

Flow_Splitter(FLOW_SPLITTER_DECLARATIONS)

Flow_Splitter(FLOW_SPLITTER_DECLARATIONS, char* _name)

void set_type(void)

This function has to be redefined in each class derived from the base Engine_Component class. It issues an error message that the component type may not be changed.

void set_type(Component_Type _type)

This function resets the component if the argument, _type, is FLOW_SPLITTER_TYPE. If it is not it issues a warning.

_type ..... *Type of engine component*

int verify_input(Text_Input** _text_input_item, Push_Button** _button)

Calls the base class **verify_input** function to check name and connection inputs and then all of the input string processing functions from this class. If any one of these error trapping functions returns REFUSED, this function returns refused.

_text_input_item ..... *Array of text input field menu items containing the new user input strings*

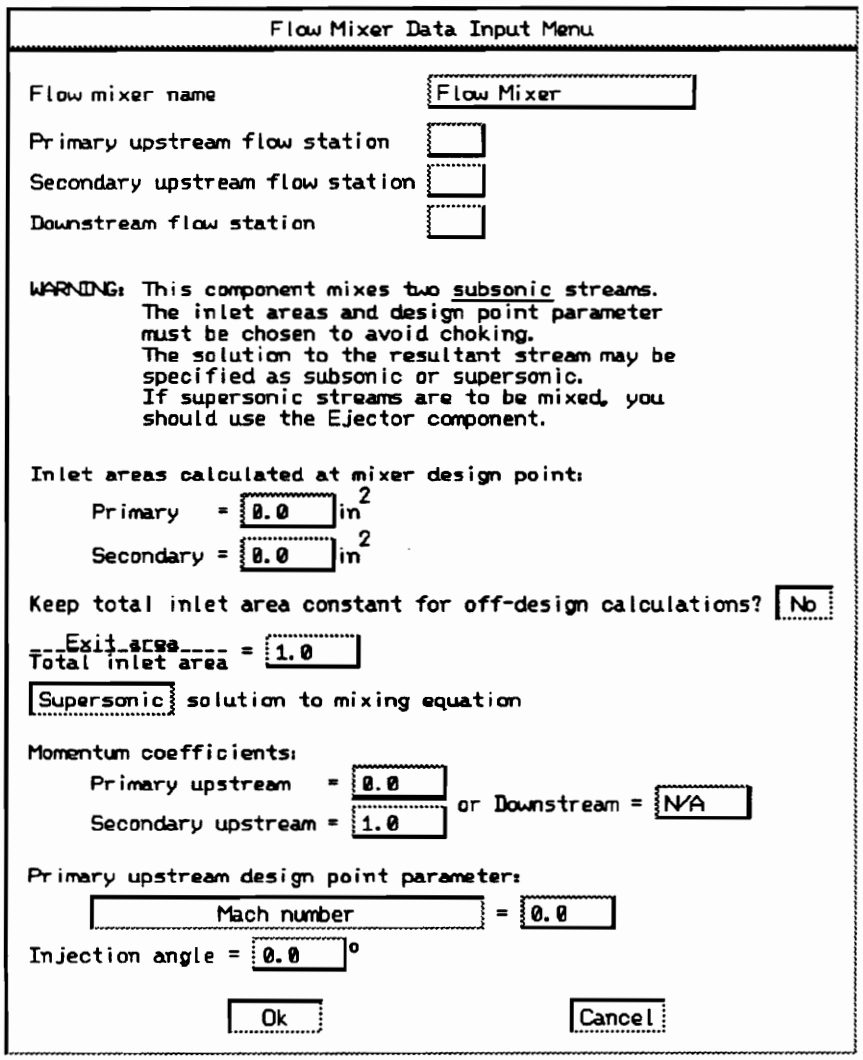
_push_button_item ..... *Array of push button menu items containing the new user specified toggle settings*

int is_flow_component(void)
Returns YES.

int is_rotational_component(void)
Returns NO.

Class Name: Flow_Mixer

Class Description: The flow mixer is a specialized engine component. The processing for flow mixer input data is defined in this class. Further error trapping, as for component/flow station connections, can be added here. The characterizing virtual functions **is_flow_component** and **is_rotational_component** are set in this class as well.



Header file Name: flow_mixer.h

Derived from: Engine_Component

Member Class: none

Member Data:

private:

int primary_area_index ..... *Index of input field for inlet area of the primary stream at design point*

int secondary_area_index ..... *Index of input field for inlet area of the secondary stream at design point*

int area_ratio_index ..... *Index of input field for the ratio of the mixer exit area to total mixer inlet area*

int primary_momentum_index ..... *Index of input field for the momentum coefficient of the primary stream - Either both primary and secondary momentum coefficients or the downstream coefficient can be specified. If not specified the field should be left blank or with "N/A".*

int secondary_momentum_index ..... *Index of input field for the momentum coefficient of the secondary stream - Either both primary and secondary momentum coefficients or the downstream coefficient can be specified. If not specified the field should be left blank or with "N/A".*

int downstream_momentum_index ..... *Index of input field for the momentum coefficient of the downstream flow - Either this variable or both primary and secondary momentum coefficients have to be specified. If not specified the field should be left blank or with "N/A".*

int mach_index ..... *Index of input field for the Mach number of the primary stream at design point - Depending on the state of the parameter type toggle button this number is used to calculate mixer inlet areas.*

int pressure_ratio_index ..... *Index of input field for the total to static pressure ratio of the primary stream at design point - Depending on the state of the parameter type toggle button this number is used to calculate mixer inlet areas.*

int injection_angle_index ..... *Index of input field for the injection angle in degrees of the primary stream*

int constant_area_index ..... *Index of toggle button specifying whether or not to keep the total mixer inlet area constant or to let the inlet areas change independently during calculation: Yes or No*

int solution_type_index ..... *Index of toggle button specifying the type of mixing solution: Subsonic or Supersonic*

int parameter_type_index ..... *Index of toggle button specifying the parameter type used for calculating the mixer inlet areas: Mach number or Total to static pressure ratio*

Member Functions:

FLOW_MIXER_DECLARATIONS is defined as the following argument list:

Graphics_Info* _graphics_info, const Connection** _connection_list_ptr, float _x, float _y

_graphics_info ..... *Object containing relevant graphics information*

_connection_list_ptr ..... *Pointer to the linked list of connections*

_x, _y ..... *Initial location of the flow mixer icon*

private:

void initializer(FLOW_MIXER_DECLARATIONS)

The initializer function for the flow mixer class sets the filename root used for all flow mixer related file input. It also initializes the name pointer arrays for upstream and downstream connections. It assigns indices into the input menu item arrays and sets the input data defaults stored in the file *.flow_mixer.defaults*.

int primary_area(char* _string)

int secondary_area(char* _string)

int area_ratio(char* _string)

int primary_momentum(char* _string)

int secondary_momentum(char* _string)

int downstream_momentum(char* _string)

int mach(char* _string)

int pressure_ratio(char* _string)

int injection_angle(char* _string)

These functions check the input string for validity as input data. They then set the NEPP data array element corresponding to the input item taking into account toggle button settings and dependencies on other input fields. The functions return REFUSED if the input string contained unacceptable data or was out of bounds.

_string ..... *This is the input string containing the user input for the input field corresponding to the function*

public:

Flow_Mixer(void)

Flow_Mixer(FLOW_MIXER_DECLARATIONS)

Flow_Mixer(FLOW_MIXER_DECLARATIONS, char* _name)

void set_type(void)

This function has to be redefined in each class derived from the base Engine_Component class. It issues an error message that the component type may not be changed.

void set_type(Component_Type _type)

This function resets the component if the argument, _type, is FLOW_MIXER_TYPE. If it is not it issues a warning.

_type ..... *Type of engine component*

void process_push_buttons(Event* _event)

Process push button events. This function toggles through push button options.

_event ..... *Event list possibly containing push button event*

int verify_input(Text_Input** _text_input_item, Push_Button** _button)

Calls the base class **verify_input** function to check name and connection inputs and then all of the input string processing functions from this class. If any one of these error trapping functions returns REFUSED, this function returns refused.

_text_input_item ..... *Array of text input field menu items containing the new user input strings*

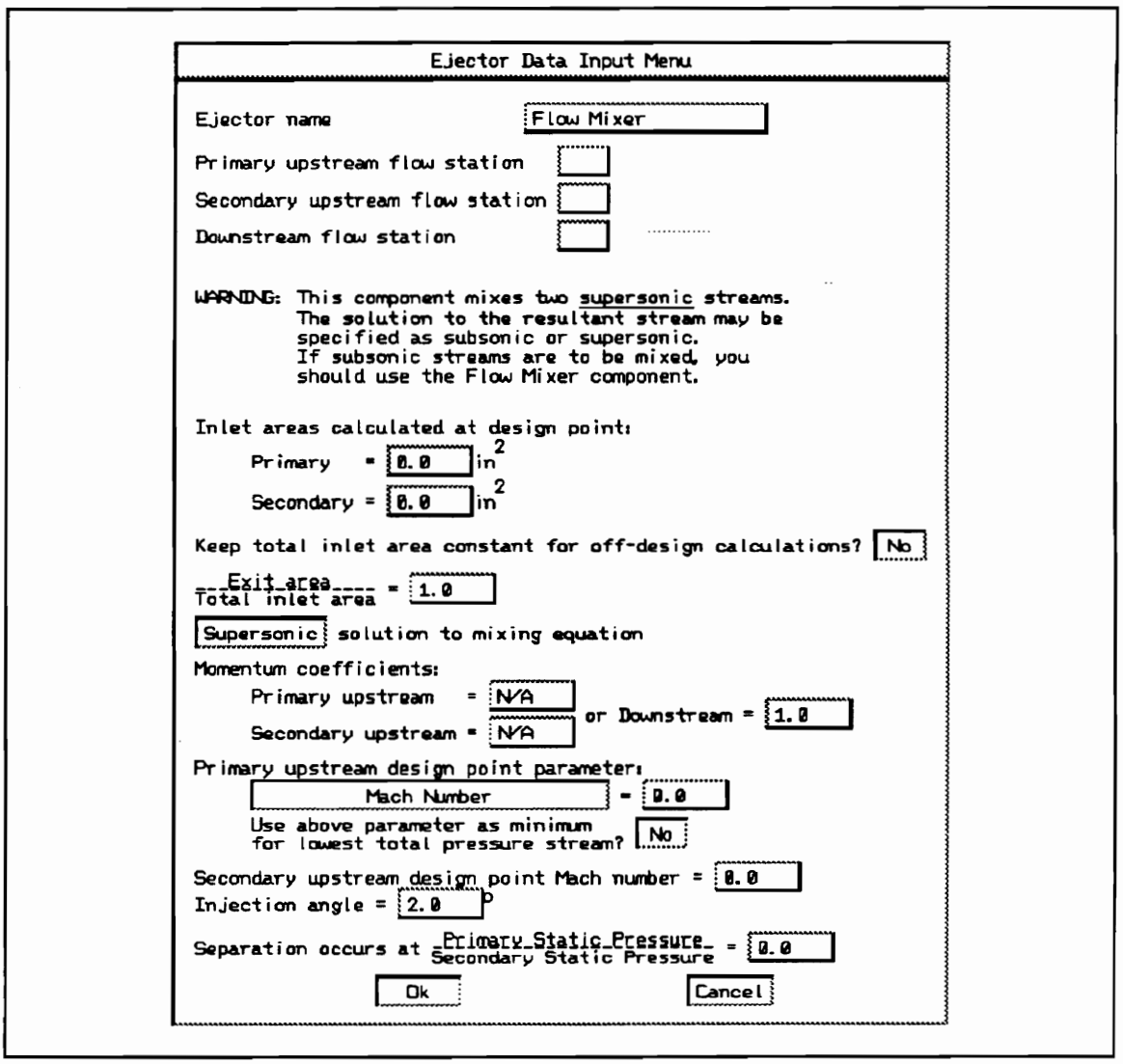
_push_button_item ..... *Array of push button menu items containing the new user specified toggle settings*

int is_flow_component(void)
Returns YES.

int is_rotational_component(void)
Returns NO.

Class Name: Ejector

Class Description: The ejector is a specialized engine component. The processing for ejector input data is defined in this class. Further error trapping, as for component/flow station connections, can be added here. The characterizing virtual functions **is_flow_component** and **is_rotational_component** are set in this class as well.



Header file Name: ejector.h

Derived from: Engine_Component

Member Class: none

Member Data:

private:

int primary_area_index ..... *Index of input field for inlet area of the primary stream at design point*

int secondary_area_index ..... *Index of input field for inlet area of the secondary stream at design point*

int area_ratio_index ..... *Index of input field for the ratio of the mixer exit area to total mixer inlet area*

int primary_momentum_index ..... *Index of input field for the momentum coefficient of the primary stream - Either both primary and secondary momentum coefficients or the downstream coefficient can be specified. If not specified the field should be left blank or with "N/A".*

int secondary_momentum_index ..... *Index of input field for the momentum coefficient of the secondary stream - Either both primary and secondary momentum coefficients or the downstream coefficient can be specified. If not specified the field should be left blank or with "N/A".*

int downstream_momentum_index ..... *Index of input field for the momentum coefficient of the downstream flow - Either this variable or both primary and secondary momentum coefficients have to be specified. If not specified the field should be left blank or with "N/A".*

int mach_index ..... *Index of input field for the Mach number of the primary stream at design point - Depending on the state of the parameter type toggle button this number is used to calculate mixer inlet areas.*

int pressure_ratio_index ..... *Index of input field for the total to static pressure ratio of the primary stream at design point - Depending on the state of the parameter type toggle button this number is used to calculate mixer inlet areas.*

int secondary_mach_index ..... *Index of input field for the Mach number of the secondary stream at design point*

int injection_angle_index ..... *Index of input field for the injection angle in degrees of the primary stream*

int separation_crit_index ..... *Index of input field for the ratio of primary stream static pressure to secondary stream static pressure -*

This is used to check for separation due to over-expansion in the ejector stream.

int constant_area_index ..... *Index of toggle button specifying whether or not to keep the total mixer inlet area constant or to let the inlet areas change independently during calculation: Yes or No*

int solution_type_index ..... *Index of toggle button specifying the type of mixing solution: Subsonic or Supersonic*

int parameter_type_index ..... *Index of toggle button specifying the parameter type used for calculating the mixer inlet areas: Mach number or Total to static pressure ratio*

Member Functions:

EJECTOR_DECLARATIONS is defined as the following argument list:

Graphics_Info* _graphics_info, const Connection** _connection_list_ptr, float _x, float _y

_graphics_info ..... *Object containing relevant graphics information*

_connection_list_ptr ..... *Pointer to the linked list of connections*

_x, _y ..... *Initial location of the ejector icon*

private:

void initializer(EJECTOR_DECLARATIONS)

The initializer function for the ejector class sets the filename root used for all ejector related file input. It also initializes the name pointer arrays for upstream and downstream connections. It assigns indices into the input menu item arrays and sets the input data defaults stored in the file *.ejector.defaults*.

int primary_area(char* _string)
int secondary_area(char* _string)
int area_ratio(char* _string)
int primary_momentum(char* _string)
int secondary_momentum(char* _string)
int downstream_momentum(char* _string)
int mach(char* _string)
int pressure_ratio(char* _string)
int secondary_mach(char* _string)
int injection_angle(char* _string)
int separation_crit(char* _string)

These functions check the input string for validity as input data. They then set the NEPP data array element corresponding to the input item taking into account toggle button settings and dependencies on other input fields. The functions return REFUSED if the input string contained unacceptable data or was out of bounds.

_string ..... *This is the input string containing the user input for the input field corresponding to the function*

public:

Ejector(void)
Ejector(EJECTOR_DECLARATIONS)
Ejector(EJECTOR_DECLARATIONS, char* _name)
void set_type(void)

This function has to be redefined in each class derived from the base Engine_Component class. It issues an error message that the component type may not be changed.

void set_type(Component_Type _type)

This function resets the component if the argument, _type, is EJECTOR_TYPE. If it is not it issues a warning.

_type.....*Type of engine component*

void process_push_buttons(Event* _event)

Process push button events. This function toggles through push button options.

_event.....*Event list possibly containing push button event*

int verify_input(Text_Input** _text_input_item, Push_Button** _button)

Calls the base class **verify_input** function to check name and connection inputs and then all of the input string processing functions from this class. If any one of these error trapping functions returns REFUSED, this function returns refused.

_text_input_item.....*Array of text input field menu items containing the new user input strings*

_push_button_item.....*Array of push button menu items containing the new user specified toggle settings*

int is_flow_component(void)

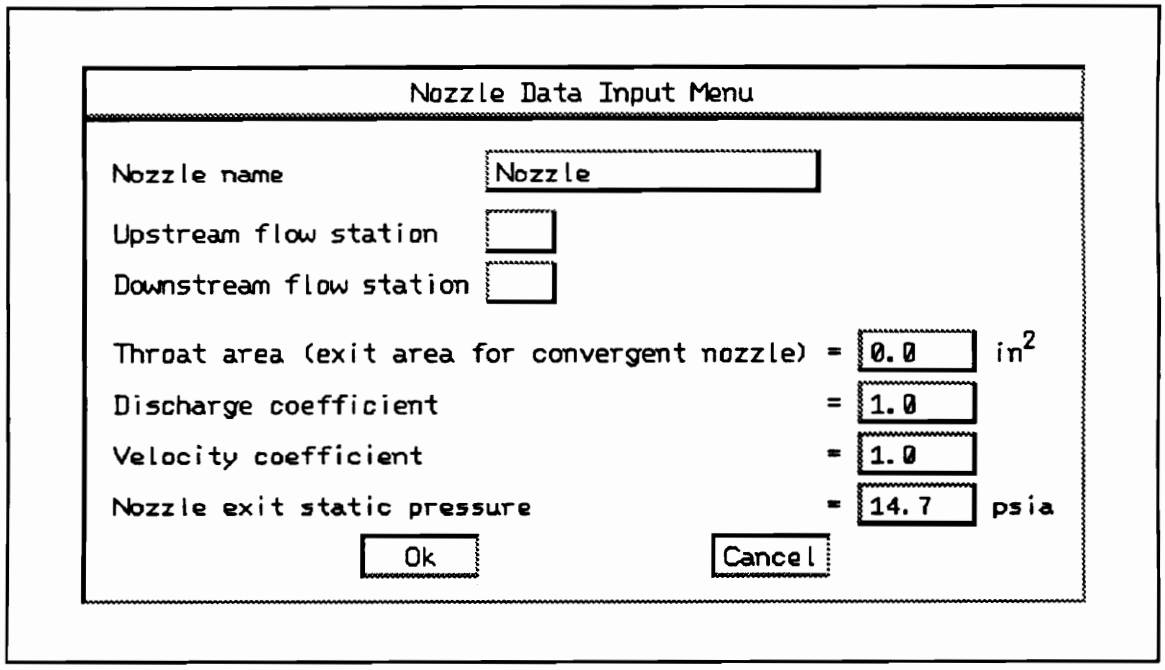
Returns YES.

int is_rotational_component(void)

Returns NO.

Class Name: Nozzle

Class Description: The nozzle is a specialized engine component. The processing for nozzle input data is defined in this class. Further error trapping, as for component/flow station connections, can be added here. The characterizing virtual functions **is_flow_component** and **is_rotational_component** are set in this class as well.



Header file Name: nozzle.h

Derived from: Engine_Component

Member Class: none

Member Data:
private:

int throat_area_index ..... *Index of input field for the nozzles throat cross-sectional area - This corresponds to the exit area for converging nozzles.*

int discharge_coefficient_index ..... *Index of input field for the nozzle discharge coefficient (C_D)*

int velocity_coefficient_index ..... *Index of input field for the velocity coefficient (C_V)*

int exit_static_pressure_index ..... *Index of input field for the nozzle exit static pressure*

Member Functions:

NOZZLE_DECLARATIONS is defined as the following argument list:

Graphics_Info* _graphics_info, const Connection** _connection_list_ptr, float _x, float _y

_graphics_info ..... *Object containing relevant graphics information*

_connection_list_ptr ..... *Pointer to the linked list of connections*

_x, _y ..... *Initial location of the nozzle icon*

private:

void initializer(NOZZLE_DECLARATIONS)

The initializer function for the nozzle class sets the filename root used for all nozzle related file input. It also initializes the name pointer arrays for upstream and downstream connections. It assigns indices into the input menu item arrays and sets the input data defaults stored in the file *.nozzle.defaults*.

int throat_area(char* _string)

int discharge_coefficient(char* _string)

int velocity_coefficient(char* _string)

int exit_static_pressure(char* _string)

These functions check the input string for validity as input data. They then set the NEPP data array element corresponding to the input item taking into account toggle button settings and dependencies on other input fields. The functions return REFUSED if the input string contained unacceptable data or was out of bounds.

_string ..... *This is the input string containing the user input for the input field corresponding to the function*

public:

Nozzle(void)

Nozzle(NOZZLE_DECLARATIONS)

Nozzle(NOZZLE_DECLARATIONS, char* _name)

void set_type(void)

This function has to be redefined in each class derived from the base Engine_Component class. It issues an error message that the component type may not be changed.

void set_type(Component_Type _type)

This function resets the component if the argument, _type, is NOZZLE_TYPE. If it is not it issues a warning.

_type ..... *Type of engine component*

void process_push_buttons(Event* _event)

Process push button events. This function toggles through push button options.

_event ..... *Event list possibly containing push button event*

int verify_input(Text_Input** _text_input_item, Push_Button** _button)

Calls the base class **verify_input** function to check name and connection inputs and then all of the input string processing functions from this class. If any one of these error trapping functions returns REFUSED, this function returns refused.

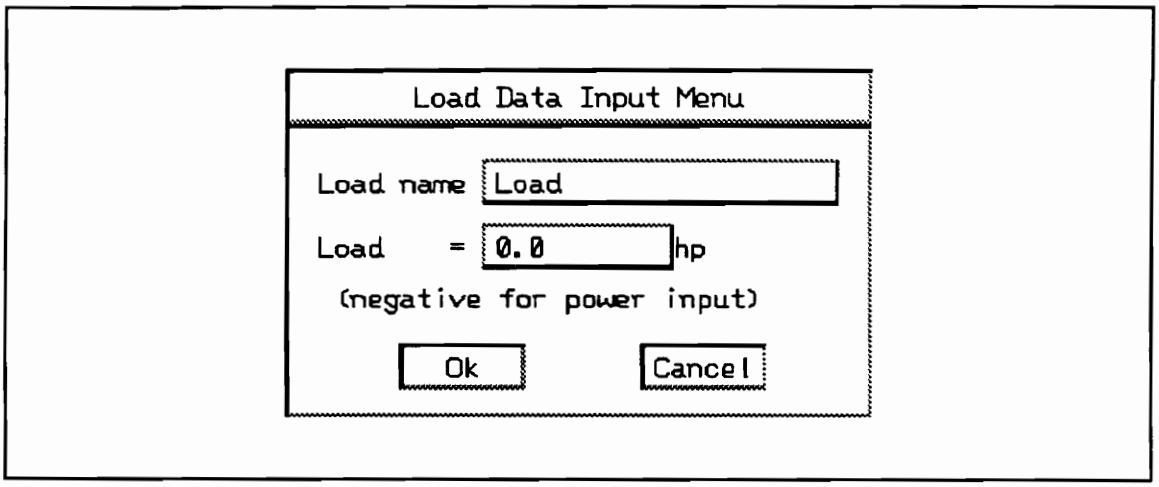
_text_input_item ..... *Array of text input field menu items containing the new user input strings*

_push_button_item.....*Array of push button menu items containing the new user specified toggle settings*

int is_flow_component(void)
Returns YES.
int is_rotational_component(void)
Returns NO.

Class Name: Load

Class Description: The load is a specialized engine component. The processing for load input data is defined in this class. Further error trapping, as for component/flow station connections, can be added here. The characterizing virtual functions **is_flow_component** and **is_rotational_component** are set in this class as well.



Header File Name: load.h

Derived from: Engine_Component

Member Class: none

Member Data:
private:
int load_index ..... ***Index of input field for the load***

Member Functions:
LOAD_DECLARATIONS is defined as the following argument list:

Graphics_Info* _graphics_info, const Connection** _connection_list_ptr, float _x, float _y

_graphics_info ..... *Object containing relevant graphics information*

_connection_list_ptr ..... *Pointer to the linked list of connections*

_x, _y ..... *Initial location of the load icon*

private:

void initializer(LOAD_DECLARATIONS)

The initializer function for the load class sets the filename root used for all load related file input. It also initializes the name pointer arrays for upstream and downstream connections. It assigns indices into the input menu item arrays and sets the input data defaults stored in the file *.load.defaults*.

int load(char* _string)

These functions check the input string for validity as input data. They then set the NEPP data array element corresponding to the input item taking into account toggle button settings and dependencies on other input fields. The functions return REFUSED if the input string contained unacceptable data or was out of bounds.

_string ..... *This is the input string containing the user input for the input field corresponding to the function*

public:

Load(void)

Load(LOAD_DECLARATIONS)

Load(LOAD_DECLARATIONS, char* _name)

void set_type(void)

This function has to be redefined in each class derived from the base Engine_Component class. It issues an error message that the component type may not be changed.

void set_type(Component_Type _type)

This function resets the component if the argument, _type, is LOAD_TYPE. If it is not it issues a warning.

_type ..... *Type of engine component*

int verify_input(Text_Input** _text_input_item, Push_Button** _button)

Calls the base class **verify_input** function to check name and connection inputs and then all of the input string processing functions from this class. If any one of these error trapping functions returns REFUSED, this function returns refused.

_text_input_item ..... *Array of text input field menu items containing the new user input strings*

_push_button_item ..... *Array of push button menu items containing the new user specified toggle settings*

int is_flow_component(void)

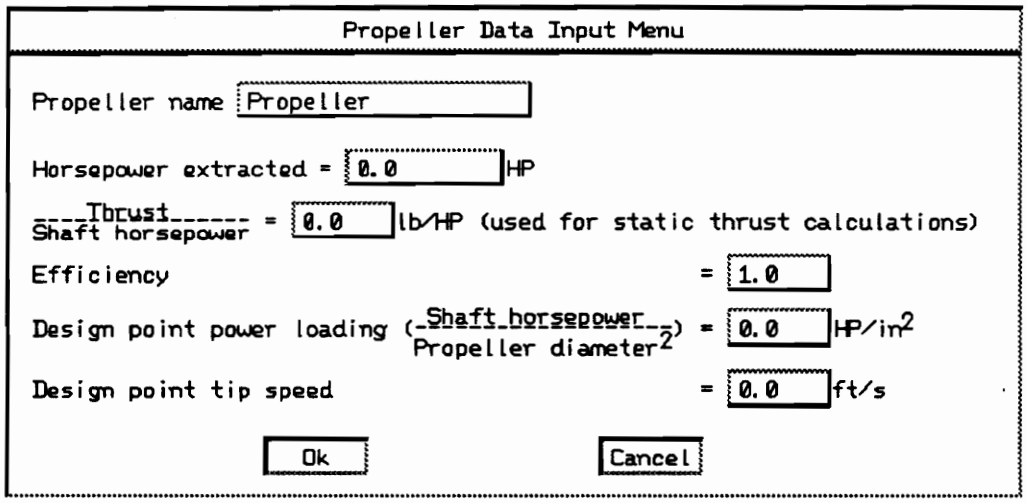
Returns NO.

int is_rotational_component(void)

Returns NO.

Class Name: Propeller

Class Description: The propeller is a specialized engine component. The processing for propeller input data is defined in this class. Further error trapping, as for component/flow station connections, can be added here. The characterizing virtual functions **is_flow_component** and **is_rotational_component** are set in this class as well.



Header File Name: propeller.h

Derived from: Engine_Component

Member Class: none

Member Data:

private:

int load_index ..... *Index of input field for the load*

int thrust_ratio_index ..... *Index of input field for the propeller thrust divided by shaft horsepower for the static thrust calculations in lb/HP*

int efficiency_index ..... *Index of input field for the propeller efficiency (Thrust*Flight Velocity/Shaft Power Input)*

int power_loading_index ..... *Index of input field for the propeller's design point power loading (Shaft HP/Propeller Diameter^2)*

int tip_speed_index ..... *Index of input field for the propeller design tip speed in ft/s*

Member Functions:

PROPELLER_DECLARATIONS is defined as the following argument list:

Graphics_Info* _graphics_info, const Connection** _connection_list_ptr, float _x, float _y

_graphics_info ..... *Object containing relevant graphics information*

_connection_list_ptr ..... *Pointer to the linked list of connections*

_x, _y ..... *Initial location of the propeller icon*

private:

void initializer(PROPELLER_DECLARATIONS)

The initializer function for the propeller class sets the filename root used for all propeller related file input. It also initializes the name pointer arrays for upstream and downstream connections. It assigns indices into the input menu item arrays and sets the input data defaults stored in the file *.propeller.defaults*.

int load(char* _string)

int thrust_ratio(char* _string)

int efficiency(char* _string)

int power_loading(char* _string)

int tip_speed(char* _string)

These functions check the input string for validity as input data. They then set the NEPP data array element corresponding to the input item taking into account toggle button settings and dependencies on other input fields. The functions return REFUSED if the input string contained unacceptable data or was out of bounds.

_string ..... *This is the input string containing the user input for the input field corresponding to the function*

public:

Propeller(void)

Propeller(PROPELLER_DECLARATIONS)

Propeller(PROPELLER_DECLARATIONS, char* _name)

void set_type(void)

This function has to be redefined in each class derived from the base Engine_Component class. It issues an error message that the component type may not be changed.

void set_type(Component_Type _type)

This function resets the component if the argument, _type, is PROPELLER_TYPE. If it is not it issues a warning.

_type ..... *Type of engine component*

int verify_input(Text_Input** _text_input_item, Push_Button** _button)

Calls the base class **verify_input** function to check name and connection inputs and then all of the input string processing functions from this class. If any one of these error trapping functions returns REFUSED, this function returns refused.

_text_input_item ..... *Array of text input field menu items containing the new user input strings*

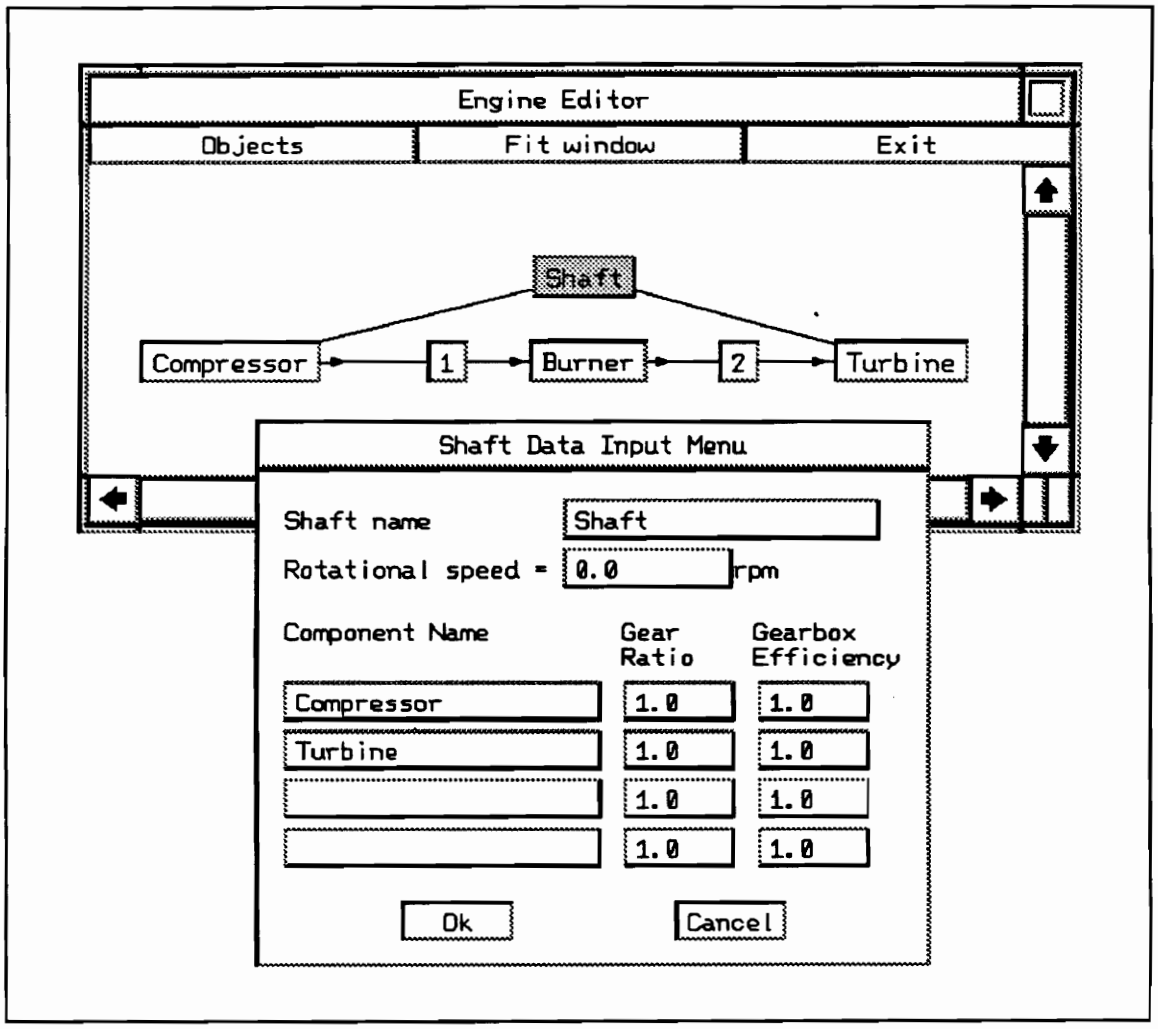
_push_button_item ..... *Array of push button menu items containing the new user specified toggle settings*

int is_flow_component(void)
Returns NO.

int is_rotational_component(void)
Returns NO.

Class Name: Shaft

Class Description: The shaft is a specialized engine component. The processing for shaft input data is defined in this class. Further error trapping, as for component/flow station connections, can be added here. The characterizing virtual functions **is_flow_component** and **is_rotational_component** are set in this class as well.



Header File Name: shaft.h

Derived from: Engine_Component

Member Class: none

Member Data:

private:

int rotational_speed_index ..... *Index of input field for the shaft rotational speed*

int gear_ratio_index[4] ..... *Index of input field for the shaft's four gear ratios to the connecting components*

int efficiency_index[4] ..... *Index of input field for the four gear box efficiencies*

Member Functions:

SHAFT_DECLARATIONS is defined as the following argument list:

Graphics_Info* _graphics_info, const Connection** _connection_list_ptr, float _x, float _y

_graphics_info ..... *Object containing relevant graphics information*

_connection_list_ptr ..... *Pointer to the linked list of connections*

_x, _y ..... *Initial location of the shaft icon*

private:

void initializer(SHAFT_DECLARATIONS)

The initializer function for the shaft class sets the filename root used for all shaft related file input. It also initializes the name pointer arrays for upstream and downstream connections. It assigns indices into the input menu item arrays and sets the input data defaults stored in the file *.shaft.defaults*.

int rotational_speed(char* _string)

int gear_ratio(int _index, char* _string)

int efficiency(int _index, char* _string)

These functions check the input string for validity as input data. They then set the NEPP data array element corresponding to the input item taking into account toggle button settings and dependencies on other input fields. The functions return REFUSED if the input string contained unacceptable data or was out of bounds.

_string ..... *This is the input string containing the user input for the input field corresponding to the function*

_index ..... *Index (1-4) into one of the four connecting component's gear ratio and gear box efficiency input fields*

public:

Shaft(void)

Shaft(SHAFT_DECLARATIONS)

Shaft(SHAFT_DECLARATIONS, char* _name)

void set_type(void)

This function has to be redefined in each class derived from the base Engine_Component class. It issues an error message that the component type may not be changed.

void set_type(Component_Type _type)

This function resets the component if the argument, _type, is SHAFT_TYPE. If it is not it issues a warning.

_type.....*Type of engine component*

int verify_input(Text_Input** _text_input_item, Push_Button** _button)

Calls the base class **verify_input** function to check name and connection inputs and then all of the input string processing functions from this class. If any one of these error trapping functions returns REFUSED, this function returns refused.

_text_input_item.....*Array of text input field menu items containing the new user input strings*

_push_button_item.....*Array of push button menu items containing the new user specified toggle settings*

int is_flow_component(void)

Returns NO.

int is_rotational_component(void)

Returns YES.

Appendix B - Detailed C Function Descriptions

Several groups of functions were used that did not warrant creation of a class. Conversion to classes may be considered at a later time. Several functions used frequently throughout the program can be found in **standard_utilities.h**. They are functions used for confirmation and message pop-up menus and also for error trapping. The file **mo_strings.h** is the header file for a collection of functions found useful for manipulating strings. Many of these functions are parsing functions used to find certain patterns in strings. These functions readily lend themselves to integration into a string class. This was not attempted due to the wide scope of the problem. The functions included with the file **mo_memory.h** were developed for memory manipulation with the use of dynamically allocated arrays.

Function Name:	**confirm**
Location:	standard_utilities.h
Description:	This function pops up a menu fitted around a one-line question and two evenly spaced push buttons. The push buttons are fitted for user provided labels.
Syntax:	*int confirm(Graphics_Info *_graphics_info, char* _question, char* _yes_choice, char* _no_choice);*
Arguments:	_graphics_info ..... Object containing relevant graphics ..... information _question ..... Question string _yes_choice..... String for the first button label _no_choice..... String for the second button label
Return Value:	If the first button was picked the function returns YES. If the second button was picked the function returns NO.

Function Name:	**message**
Location:	standard_utilities.h
Description:	This function pops up a menu fitted around a one-line message and a centered push button. The push button are fitted for a user provided label.
Syntax:	void message(Graphics_Info *_graphics_info, char* _message, char* _ok);

Arguments:
_graphics_info ..... Object containing relevant graphics
..... information
_message ..... Message string
_ok..... String for the button label

Return Value: void

Function Name: **check_number_input**
Location: standard_utilities.h
Description: This function parses a text string for valid number input. It gets checked against maximum, minimum, the existence of a number in the input string and, if so specified, the existence of the word "MAP", "Map", "map" or "N/A" in the string. If any valid input value is found it is returned as a float in the argument, _number. If no valid value was found the argument, _number, gets set to the default value, _default. If maximum or minimum is exceeded a message is popped up to that effect and the number is set to the maximum or minimum. If "MAP", "Map" or "map" is found and the _map_flag was not set, an error message is popped up.

Syntax: int check_number_input(Graphics_Info* _graphics_info, float* _number, char** _text, const char* _input_name, float _max, float _min, float _default, int _map_flag);

Arguments:
_graphics_info ..... Object containing relevant graphics
..... information
_number..... The function uses this argument to return
..... the processed value of the input.
_text..... String containing the input
_input_name..... String containing the name of the variable
..... being checked - This is used in error
..... messages
_max..... The maximum allowed value of the variable
_min ..... The minimum allowed value of the variable
_default..... The default value of the variable
_map_flag..... Is the input "MAP", "Map" or "map"
..... allowable as valid input? - YES or NO

Return Value: If the string does not contain a valid value the function returns REFUSED.
If the value found exceeds the maximum the function returns REFUSED.
If the value found is less than the minimum the function returns REFUSED.
If the string found was "MAP", "Map" or "map" the function returns MAP.
If the string found was "N/A" the function returns NA.

Function Name: **new_strcpy**
Location: mo_strings.h
Description: This function combines C++'s **new** operator and the standard **strcpy** function. It reserves the memory required to store the character string, _source. It then copies _source to that location and assigns the address of that location to _destination.

Syntax: void new_strcpy(char** _destination, const char* _source);

Arguments:
_destination ..... A pointer to a string - This pointer will be
..... used to point to the newly copied string
_source ..... String to be copied

Return Value: void

Function Name: **strip**
Location: mo_strings.h
Description: This function strips leading and trailing blanks from a string.
Syntax: char* strip(const char* _source);
Arguments: _source ..... String to be copied
Return Value: Returns the stripped string

Function Name: **get_next_label**
Location: mo_strings.h
Description: This function returns the first non-format portion of a string. A format
starts with a ..... '%'
contains an optional signed float... [+|-][ddd][.][ddd]
and ends with a letter ..... [a..zA..Z]
Two percent signs, '%%', are interpreted as a single '%' without special formatting.
EXAMPLE:
get_next_label("%-1.3f this is a label %T Got it?");
== " this is a label "
Syntax: char * get_next_label(const char* _source);
Arguments: _source ..... String to be parsed
Return Value: Returns the first label portion of _source.

Function Name: **get_next_format_value**
Location: mo_strings.h
Description: This function scans the string _source for the first format and returns its value.
A format starts with a..... '%'
contains an optional signed float... [+|-][ddd][.][ddd]
and ends with a letter ..... [a..zA..Z]
Syntax: float get_next_format_value(const char *_source);
Arguments: _source ..... String to be parsed
Return Value: If no value was found the function returns 1.0.
If a value was found the function returns that value as a float.

Function Name: **find_next_format**
Location: mo_strings.h
Description: This function parses the string, _source, for the starting position of the next format. Formats preceded by a second '%' are ignored.
Syntax: int find_next_format(const char* _source);
Arguments: _source ..... String to be parsed
Return Value: Returns the starting position of the next format

Function Name: **find_next_escape**
Location: mo_strings.h
Description: This function parses the string, _source, for the starting position of the next escape sequence. Escape sequences start with a '\' and end with a letter. Duplicate backslashes, '\\' are interpreted as '\' and ignored.

Syntax: int find_next_escape(const char* _source);
Arguments: _source ..... String to be parsed
Return Value: Returns the starting position of the next escape sequence.

Function Name: **find_next_format**
Location: mo_strings.h
Description: This function parses the string, _source, for the starting position of the next format of a type _format_type.
The format starts with a ..... '%'
contains an optional signed float ... [+|-][ddd][.][ddd]
and ends with a character..... _format_type
Formats preceded by a second '%' are ignored.
Syntax: int find_next_format(const char _format_type, const char* _source);
Arguments: _format_type..... Character terminating the format
_source ..... String to be parsed
Return Value: Returns the starting position of the next format of type _format_type

Function Name: **find_next_escape**
Location: mo_strings.h
Description: This function parses the string, _source, for the starting position of the next escape sequence terminated by the character _escape_type. Escape sequences start with a '\' and end with a letter. Duplicate backslashes, '\\' are interpreted as '\' and ignored.
Syntax: int find_next_escape(const char _escape_type, const char* _source);
Arguments: _escape_type..... Character terminating the escape sequence
_source ..... String to be parsed
Return Value: Returns the starting position of the next escape sequence terminated by _escape_type.

Function Name: **find_next_label**
Location: mo_strings.h
Description: This function parses the string, _source, for the position of the next label (text not containing a format or escape sequence)
Syntax: int find_next_label(const char* _source);
Arguments: _source ..... String to be parsed
Return Value: Returns position of the first label in the string.

Function Name: **remove_next_format**
Location: mo_strings.h
Description: This function removes the next format from a character string, _source. If no format is found, no change is made. Extra memory is NOT freed.
Syntax: void remove_next_format(char* _source);
Arguments: _source ..... String to be parsed
Return Value: void

Function Name: **remove_next_escape**
Location: mo_strings.h

Description: This function removes the next escape sequence from a character string, _source. If no escape sequence is found, no change is made. Extra memory is NOT freed.
Syntax: void remove_next_escape(char* _source);
Arguments: _source ..... String to be parsed
Return Value: void

Function Name: **remove_next_label**
Location: mo_strings.h
Description: This function removes the next label from a character string, _source. If no label is found, no change is made. Extra memory is NOT freed.
Syntax: void remove_next_label(char* _source);
Arguments: _source ..... String to be parsed
Return Value: void

Function Name: **append_string**
Location: mo_strings.h
Description: This function allocations the memory required to store the combination of a destination string and a source string. _destination and _source are concatenated to the reserved memory location and reassigned to _destination.
Syntax: void append_string(char** _destination, const char* _source);
Arguments: _destination ..... A pointer to a string - This pointer points ..... to the front part of the new string and will ..... be used to point to the newly appended ..... string
_source ..... String to be appended to _destination
Return Value: void

Function Name: **find_next_exp_float_string**
Location: mo_strings.h
Description: This function finds the next portion of the string _source, to fit the format of a float with an optional sign, optional decimal point, and optional exponential notation: [+|-][ddd][.][ddd][eIE[+|-]ddd].
Syntax: int find_next_exp_float_string(char* _source);
Arguments: _source ..... String to be parsed
Return Value: Returns the starting position of the number

Function Name: **get_next_exp_float_string**
Location: mo_strings.h
Description: This function returns the next portion of the string _source, to fit the format of a float with an optional, sign, optional decimal point, and optional exponential notation: [+|-][ddd][.][ddd][eIE[+|-]ddd].
Syntax: char* get_next_exp_float_string(char* _source);
Arguments: _source ..... String to be parsed
Return Value: Returns the number in string format

Function Name: **find_next_float_string**
Location: mo_strings.h

Description: This function finds the next portion of the string _source, to fit the format of a float with an optional sign and optional decimal point: [+|-][ddd][.][ddd].
Syntax: int find_next_float_string(char* _source);
Arguments: _source ..... String to be parsed
Return Value: Returns the starting position of the number

Function Name: **get_next_float_string**
Location: mo_strings.h
Description: This function returns the next portion of the string _source, to fit the format of a float with an optional sign and optional decimal point: [+|-][ddd][.][ddd].
Syntax: char* get_next_float_string(char* _source);
Arguments: _source ..... String to be parsed
Return Value: Returns the number in string format

Function Name: **find_next_int_string**
Location: mo_strings.h
Description: This function finds the next portion of the string _source, to fit the format of an integer with an optional sign: [+|-][ddd].
Syntax: int find_next_int_string(char* _source);
Arguments: _source ..... String to be parsed
Return Value: Returns the starting position of the number

Function Name: **get_next_int_string**
Location: mo_strings.h
Description: This function returns the next portion of the string _source, to fit the format of an integer with an optional sign: [+|-][ddd].
Syntax: char* get_next_int_string(char* _source);
Arguments: _source ..... String to be parsed
Return Value: Returns the number in string format

Function Name: **get_string_array_from_file**
Location: mo_strings.h
Description: This function reads an array of strings into _destination from a file, _source, and counts the number of lines read into _number_of_lines.
Syntax: int get_string_array_from_file(char*** _destination, int* _number_of_lines, const char* _source);
Arguments: _destination ..... Pointer to an array of strings - This pointer
..... will be used to point to the array of read
..... strings.
_number_of_lines ..... Number of lines read
_source ..... Name of the file to be read in
Return Value: If the file cannot be opened the return value is 0.
If any lines were read it is 1.

Function Name: **crop_string_to_length**
Location: mo_strings.h

Description: This function crops characters from a string, _source, from left, right or both sides until its PHIGS representation fits into a certain space, _length.

Syntax: char* crop_string_to_length(const char* _source, float _character_height, int _font, float _length, int _alignment);

Arguments:
_source ..... String to be cropped
_character_height ..... PHIGS text character height attribute
_font ..... PHIGS text font attribute
_length..... Space to fit string into
_alignment..... Alignment of text - If text is LEFT aligned
..... it will be cropped on the right. If text is
..... RIGHT aligned it will be cropped on the
..... left. If it is CENTER aligned it will be
..... cropped on both sides if necessary.

Return Value: Returns the cropped character string

Function Name: **crop_string_to_length**
Location: mo_strings.h
Description: This function crops characters from a string, _source, from both sides until its PHIGS representation fits into a certain space, _length.

Syntax: char* crop_string_to_length(const char* _source, float _character_height, int _font, float _length);

Arguments:
_source ..... String to be cropped
_character_height ..... PHIGS text character height attribute
_font ..... PHIGS text font attribute
_length..... Space to fit string into

Return Value: Returns the cropped character string

Function Name: **append_array**
Location: mo_memory.h
Description: This function reallocates memory for an array of integers, _array, to accommodate another element, _element.

Syntax: void append_array(int** _array, const int _element, const int _element_index);

Arguments:
_array ..... Pointer to an array of integers - This pointer
..... points to the source array and will point to
..... the destination array at the end of the
..... function.
_element ..... New array element
_element_index..... New position of the last array element -
..... Also, the total number of array elements

Return Value: void

Function Name: **append_array**
Location: mo_memory.h
Description: This function reallocates memory for an array of floats, _array, to accommodate another element, _element.

Syntax: void append_array(float** _array, const float _element, const int _element_index);

Arguments: _array ..... Pointer to an array of floats - This pointer
..... points to the source array and will point to
..... the destination array at the end of the
..... function.
_element ..... New array element
_element_index..... New position of the last array element -
..... Also, the total number of array elements

Return Value: void

Function Name: **append_array**
Location: mo_memory.h
Description: This function reallocates memory for an array of characters, _array, to accommodate another element, _element.
Syntax: void append_array(char** _array, const char _element,
const int _element_index);
Arguments: _array ..... Pointer to an array of character - This
..... pointer points to the source array and will
..... point to the destination array at the end of
..... the function.
_element ..... New array element
_element_index..... New position of the last array element -
..... Also, the total number of array elements

Return Value: void

Function Name: **append_array**
Location: mo_memory.h
Description: This function reallocates memory for an array of strings, _array, to accommodate another element, _element.
Syntax: void append_array(char*** _array, const char* _element,
const int _element_index);
Arguments: _array ..... Pointer to an array of strings - This pointer
..... points to the source array and will point to
..... the destination array at the end of the
..... function.
_element ..... New array element
_element_index..... New position of the last array element -
..... Also, the total number of array elements

Return Value: void

Function Name: **append_array**
Location: mo_memory.h
Description: This function reallocates memory for an array of strings, _array, to accommodate another element, _element. It identifies the end of the current array if it finds a pointer to NULL.
Syntax: void append_array(int** _array, const int _element);
Arguments: _array ..... Pointer to an array of strings - This pointer
..... points to the source array and will point to
..... the destination array at the end of the

..... function.
_element ..... New array element

Return Value: void

Function Name: **append_array**
Location: mo_memory.h
Description: This function reallocates memory for an array of pointers to strings, _array, to accommodate another element, _element.
Syntax: void append_array(const char* const*** _array, const char* const* _element, const int _element_index);
Arguments:
_array ..... Pointer to an array of pointers to strings -
..... This pointer points to the source array and
..... will point to the destination array at the end
..... of the function.
_element ..... New array element
_element_index ..... New position of the last array element -
..... Also, the total number of array elements

Return Value: void

Function Name: **append_array**
Location: mo_memory.h
Description: This function reallocates memory for an array of pointers to strings, _array, to accommodate another element, _element. It identifies the end of the current array if it finds a pointer to NULL.
Syntax: void append_array(const char* const*** _array,
const char* const* _element);
Arguments:
_array ..... Pointer to an array of pointers to strings -
..... This pointer points to the source array and
..... will point to the destination array at the end
..... of the function.
_element ..... New array element

Return Value: void

Function Name: **append_array**
Location: mo_memory.h
Description: This function reallocates memory for an array of characters, _array, to accommodate appending a string, _string.
Syntax: void append_array(char** _array, const char* _string,
const int _element_index);
Arguments:
_array ..... Pointer to an array of character - This
..... pointer points to the source array and will
..... point to the destination array at the end of
..... the function.
_string ..... String of characters to be appended to
..... _array
_element_index ..... New position of the last array element -
..... Also, the total number of array elements

Return Value: void

Function Name: **copy_array**
Location: mo_memory.h
Description: This function allocates memory to copy an array of strings, _array, to a destination, _destination.
Syntax: void copy_array(char*** _destination, const char** _source);
Arguments:
_destination ..... Pointer to an array of strings - This
..... pointer will point to the destination array at
..... the end of the function.
_source ..... String array to be copied
Return Value: void

Function Name: **insert_array_element**
Location: mo_memory.h
Description: This function reallocates memory for an array of strings, _array, to accommodate another element, _element, at a specified position in the array.
Syntax: void insert_array_element(char*** _array, const char* _element, const int _element_index);
Arguments:
_array ..... Pointer to an array of strings - This pointer
..... points to the source array and will point to
..... the destination array at the end of the
..... function.
_element ..... New array element
_element_index..... Position of the new element in the array
Return Value: void

Function Name: **pop_array**
Location: mo_memory.h
Description: This function removes the first element of an array of strings and returns it. The function frees excess memory by reallocating the new array.
Syntax: char* pop_array(char*** _array);
Arguments:
_array ..... Pointer to an array of strings - This pointer
..... points to the source array and will point to
..... the destination array at the end of the
..... function.
Return Value: Returns popped string

Function Name: **delete_array**
Location: mo_memory.h
Description: This function frees the memory used by an array of character strings.
Syntax: void insert_array_element(char*** _array);
Arguments:
_array ..... Pointer to an array of strings - This pointer
..... points to the source array and will point to
..... the destination array at the end of the
..... function.
Return Value: void

Vita

Andreas Steude was born on October 6, 1967 in Leverkusen, West Germany. At the age of 12 his family was transplanted to Pittsburgh, Pennsylvania, USA, where he graduated from Mt. Lebanon High School in 1985. The author has always admired the aesthetics of elegant efficiency. The sight of a glider in flight or even the workings of intricate machinery fascinated him. Thus, with a strong interest in mathematics and the physical sciences, he began the study of aerospace engineering at Virginia Tech. With an emphasis on aerodynamics and, later, aircraft propulsion, he continued to graduate school. Soon after graduating with his Master of Science degree from the Department of Mechanical Engineering at Virginia Tech the author hopes to become a citizen of the United States of America.