

# Project Integration Architecture: Application Architecture

*Dr. William Henry Jones*

National Aeronautics and Space Administration  
John H. Glenn Research Center at Lewis Field  
Cleveland, OH 44135  
216-433-5862  
William.H.Jones@grc.nasa.gov

X00.00	22	Oct	1998
X00.01	08	Dec	1998
X00.02	29	Mar	1999
X00.03	24	Aug	2000
X00.04	28	Aug	2000
X00.05	19	Sep	2000
X00.06	21	Jun	2002

Keywords:

PIA; PRICE; CORBA; Knowledge Management; Application Integration;  
Technical Integration Technologies; Semantic Encapsulation; Self-Revelation;

**ABSTRACT:** *The Project Integration Architecture (PIA) implements a flexible, object-oriented, wrapping architecture which encapsulates all of the information associated with engineering applications. The architecture allows the progress of a project to be tracked and documented in its entirety. Additionally, by bringing all of the information sources and sinks of a project into a single architectural space, the ability to transport information between those applications is enabled.*

## 1 Introduction

In the late 1980's, the Integrated CFD and Experiments (ICE) project [1, 2] was carried out with the goal of providing a single, graphical user interface (GUI) and data management environment for a variety of CFD codes and related experimental data. The intent of the ICE project was to ease the difficulties of interacting with and intermingling these disparate information sources. The project was a success on a research basis; however, due to various technical limitations (for instance, the difficulty of developing object-oriented constructs in a non-object-oriented language) and the loss of key personnel, it was deemed inappropriate to advance the effort beyond the successes achieved to that point. Thus, a re-engineering of the project was initiated in 1996. The effort was first renamed Portable, Redesigned Integrated CFD and Experiments (PRICE) project and then, as the wide applicability of the concepts came to be appreciated, Project Integration Architecture (PIA).

Two key re-engineering decisions were made: the C lan-

guage used by the ICE project would be abandoned in favor of the now-available C++ object-oriented extension to that language and the graphical user interface would be eliminated as a product element of the project. The first decision was but a matter of circumstances; had C++ been available to the original ICE team at project outset, it would almost certainly have been selected for use. The second decision, to remove the GUI from the project product set, was taken only after a period of time and reflected two truths: first, that a cross-platform GUI with the scope of functionality envisioned was far beyond the resources of the PIA project and, second, that such a GUI was duplicative of other efforts within the Agency.

During the intervening years, work has proceeded and an operational demonstration of the PIA project has been achieved. The current effort has not achieved the complete range of functionality developed in the original ICE; however, the architectural dividing line has been more thoroughly defined and adherence to it has been more rigorous. Portability of the architecture, not only across platforms, but to distributed object architectures, has been demon-

strated. This path ahead is more clearly defined.

## 2 Goals

Put in simple terms, the basic goal of the PIA effort is to capture in its entirety the usage of any technical application in a single, useful, well-defined form. This capturing is not limited to the simple output of the application itself, but further includes coordinating information and the technologist's own insights into the meaning of the information.

The nature of a 'technical application' is, by project design, nebulous: it is considered to be any computer-realized 'thing' which provides or generates useful information about a project. This may include geometry definitions extracted from Computer Aided Design (CAD) programs, geometry or other specifications derived from design code predictions, results of experimental investigations, analysis and simulation, and more.

By bringing all of the information of the technical process into one architectural design, a number of advantages are expected to (and, it is believed, do) accrue.

1. The information about the information (referred to in some conceptions as the meta-data) is encapsulated with the information itself. Information about the conditions at which the information was generated, or the merit of the information, is no longer separately stored in a technologist's unlocatable journal.
2. A common tool set for the use of applications is possible. A single GUI with a single look-and-feel can be devised so as to reduce the technologist's learning curve for additional applications to that solely related to the application. The technologist's habits of exploring a problem can now be the same from one application to the next.
3. Common browsers and search engines may be implemented to peruse the supply of information in detail and convert it into information in general for end consumers of that knowledge.
4. By wrapping every application in a well-defined architecture, it is now possible to code into such applications the knowledge to acquire information automatically from other applications. Because of the architectural design, such coded knowledge is based upon the kind of information desired, rather than upon the application generating that kind of information.
5. Wrapped applications coded to obtain information based upon its kind may then be combined in a di-

rected application graph to build, in effect, super-applications. Applications of differing fidelities and disciplines may be mixed together as appropriate to the project.

6. The building of super-applications enables project-wide optimization and/or sensitivity analysis to be conducted.

## 3 Self Revelation

Perhaps the key technology that enables the goals above is that of self revelation, the ability of a thing to reveal to others its own nature. Such a capacity can be implemented by many different techniques; however, this capacity is a very natural element of object-oriented technology.

The concept of self revelation as discussed in the two sections below can be very quickly understood by the simple analogy of meeting a new person at a party. One of the natural things to do in such a situation is to ask what the person does; if that person answers, for example, that she is a medical doctor, an entire course may be set based upon the inquisitor's needs. If the inquisitor has a medical condition for which he desires a free opinion, it may be appropriate to inquire further as to the doctor's specialty. On the other hand, if the inquisitor has not the first interest in medical topics, it may be time to discretely spill something and hurry off for a napkin.

To see the importance of this concept, consider the analogous alternative: what if one did not have the ability to inquire of new people met at a party? In that event one would be either non-functional as a party animal, or parties would have to be incredibly rigid in their formulation so as to meet the practical breadth of programmed expectation in the participants. Either no conversations could be permitted because the ability to predict the relevance would be entirely lacking, or all guest-to-guest interactions would have to be predetermined and pre-scripted so as to assure that one talked with others about relevant subjects. In either case, a lack of self revelation would be quickly seen as truly confining.

### 3.1 Self Revelation of Kind

The revelation of kind identifies the essential character of the revealing entity. In the object-oriented implementation of PIA, this sets expectations as to the kind of information and functionality a particular object has: it is a free-stream Mach number parameter, or it is an operation that may or may not be enabled and, if enabled, will execute

and do something, or it is an application offering parameters, identifications, and operations, or it is any one of an almost unlimited variety of other things. The key feature is that, upon finding out its kind, precise expectations as to what it has and what it is willing to do may be confidently inferred.

The revelation of kind is effected in two ways: an interrogative form and a declarative form. In the interrogative form, a predicate of kind is posed to the object and either affirmed or denied. In the declarative form, an inquiry is made of the object and a simple coded value is returned declaring the type of the object.

Because of the derivational nature of object technology, both of these revelational forms support the concept of depth. That is, an object may be of a particular kind at some derivational depth but, because of further derivation, may not appear to be of that kind on its surface. The examination of such layers of meaning is referred to in the PIA nomenclature as ecdysiastical analysis (from the Greek *ekdysis*, *ekdyein*, to get out of, strip off).

### 3.2 Self Revelation of Content

The revelation of content identifies the extent to which expectations based upon the revelation of kind are, in fact, fulfilled. Here, the revealed kind of an object allows one to expect that it has content of a given nature, but that nature may yet be nebulous by design, or may be variable in its amount, or may be variable in other specified ways.

Consider for example an application object. As will be discussed shortly, an application object is known to have a set of operations encapsulated in operation objects; however, by specification, it is not known whether or not there actually are any operation objects (that is, the set could be null), nor if there are any such operation objects, precisely which kinds will be present. Code consuming an application object must deal with it on that basis; that while an operation object set is defined, it may be empty and, if it is not empty, that further interrogation of individual operation objects will be necessary to determine the nature of the operations available.

## 4 Application Architecture

Building upon the concept of self revelation, an application architecture as depicted in Figure 4.1 has been devised. While the structure may, at first, appear daunting, it is, in fact, a quite orderly thing which may be easily understood.

An application presented in the image of PIA begins with a central application object, labeled **PacAppl** in the upper center of the figure, which is the root structure from which all further components emanate. Four principal components are currently provided by the **PacAppl** object:

1. A set of operations that the application is willing to perform,
2. A mass of data which the application currently contains,
3. A structure by which the contained data is identified, and
4. An ecdysiastical sorting of the information-bearing objects in the application.

The first three components are depicted in the figure in the upper left, the upper central to lower left diagonal, and the central right, respectively. The fourth component is not depicted due to its structural complexity. Each of these components is taken up in its own subsection below.

### 4.1 Application Components

#### 4.1.1 Parameter Configurations

The architectural discussion begins with the holders of parameters (the objects which hold application data of all forms) depicted as the structure proceeding to the lower left from the **PacAppl** object. These objects, labeled **PacCfg** in the figure, are called ‘configurations’. If one considers the aggregate of all data in an application (both input and output of all types and forms) to constitute an  $n$ -dimensional space (where, admittedly,  $n$  can be quite a large number), then a parameter configuration is considered to identify exactly one point in that dimensional space. Put less formally, a configuration is simply a distinct set of input data and (as appropriate) the output data it gives rise to.

Because of this definition, a changed data value constitutes a new data configuration and, commonly, results in a new **PacCfg** object. Because the data set of a typical PIA-wrapped application is expected to be large, it was decided that simple replication of the entire data set each time a new configuration occurred would be wasteful. Thus, as depicted in the figure, **PacCfg** objects are arranged in an  $n$ -ary tree and the **PacAppl** object identifies the **PacCfg** acting as the patriarch of that tree. Descendent **PacCfg** objects are considered to inherit missing data components from their ancestral configurations, thus eliminating the need to replicate unaltered data.

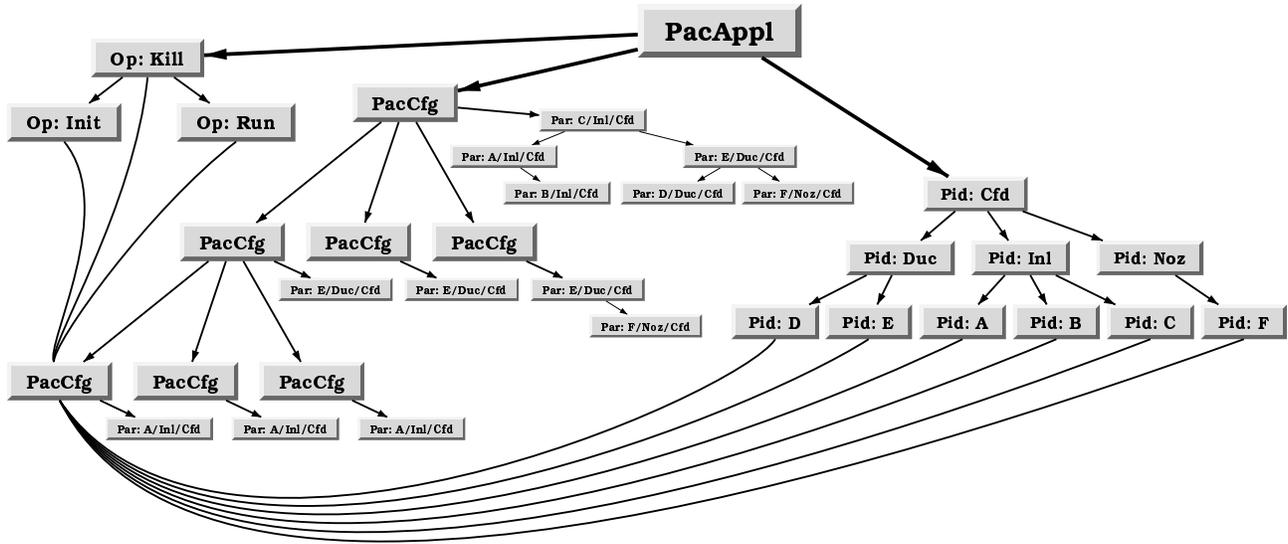


Figure 4.1: PIA Application Architecture

The potential operation of this architecture can be seen in the figure. The **PacCfg** object acting as the patriarch of the configuration tree (that is, the **PacCfg** object directly pointed to by the **PacAppl** object) will often contain the fully populated data set of the problem being explored. In the figure this would be the parameter objects labeled A/Inl/Cfd through F/Noz/Cfd. Descendent **PacCfg** objects would contain only the parameter objects being changed in the course of research investigation. In the figure, the first two direct descendents of the patriarch change only the E/Duc/Cfd parameter object while the third direct descendent also changes the F/Noz/Cfd parameter object. The three further descendents of the first direct descendent go on to hold modified values of the A/Inl/Cfd parameter object. The bottom leftmost configuration thus has its own A/Inl/Cfd parameter object, inherits its E/Duc/Cfd item from its direct parent, and inherits all remaining parameter objects of the comprehensive set from the patriarch.

Parameter objects within a configuration are maintained as a map (implemented in fact as a balanced, binary tree) sorted by a fully-qualified name (for example, the A/Inl/Cfd name above). Duplicate names are not allowed, so each parameter object must have a unique, fully-qualified name. The reason for this arrangement is to avoid the needless replication of data structure in cases where only that structure uniquely identifies a particular parameter object to be held in modified form in a dependent configuration. The structure of data is moved into the content of the fully-qualified name so that only that particular parameter object need be replicated (in modified form) in a descendent configuration.

To make this fully-qualified name concept more concrete, consider a multi-block CFD code in which each block has a set of repeated attributes whose particular value may vary from block to block. Thus, block 1 would have the attributes A, B, and C, as would block 2, block 3, and so on. Without the movement of structure into the fully-qualified name, it would be at least necessary to replicate blocks 1, 2, and 3 in order to change the B attribute of block 3 so as to make it clear that the modified B attribute is, in fact, the B attribute of the third block. By moving structure into the fully-qualified name, perhaps by naming the attribute B/Block3/SomeCfdCode, it is only necessary to replicate the modified B attribute in the descendent configuration. The fact that it is the B attribute of the third block is made clear by the fully-qualified name.

By expanding the configuration tree as work progresses, a researcher may parametrically explore a design space with reasonable economy while leaving a comprehensive documenting trail behind. As will be discussed in a later section, a number of distinct policies may be implemented with regard to the establishment of a new configuration as opposed to the modification of an existing configuration.

#### 4.1.2 Parameter Identification

The configuration structure discussed above introduced the concept of the fully-qualified name whose purpose was to capture the structure of data without requiring the needless replication of that structure within each parameter configuration. The revelation of structure within the data (which is

a revelation of content for the **PacAppl** object) is taken up by the identification structure depicted at the central right of the Figure 4.1. These are the objects whose labels begin **Pid**, each of which is followed by a name text.

This parameter identification element of the application architecture is, again, arranged as an  $n$ -ary tree. In this case, the tree structure is used to reveal the corresponding data structure of the application. The fully-qualified name of a parameter is developed by concatenating the names of each of the tree elements leading to the final identification of that parameter. In the example of the figure, all data proceeds from the application root, Cfd. Cfd has three major data structures: Duc (Duct), Inl (Inlet), and Noz (Nozzle). Descending from this level, Inl has three parameters, A, B, and C, Duc has two parameters, D and E, and Noz has one parameter, F. Thus, the fully-qualified name of the B parameter obtained by concatenating the names of the path elements leading to its identification would be B/Inl/Cfd.

Beyond this point, the parameter identification structure is unremarkable. The only point to be made is that, at any particular level of the tree, the names of identified data items/structures must be unique. Thus, the Noz identification cannot be repeated at its level. Shifting to the example of the multi-block CFD code, this means that the name 'Block' may not simply be repeated. Instead, names such as 'Block01', 'Block02', 'Block03' and so on must be generated (probably as a dynamic response to problem setup) to make the multiple-block level of the data structure unambiguous.

### 4.1.3 Operations

A key element of most applications is not solely that they hold data, but that they do something with that data. Often some algorithm is executed during which inputs are turned into outputs. To reveal these operations, the **PacAppl** object identifies a map of operational objects (labeled **Op** followed by a name in upper left of Figure 4.1) sorted by operation name, which must be unique. In the example of the figure, three operations, Init (Initialize), Kill (Kill a running operation), and Run (take the input data and run the operation to completion, acquiring new output data), are provided.

Some operations may require specific interaction with the researcher. For instance, an operation obtaining input from a file may need to prompt the researcher to identify the file to be used. For this purpose, a GUI call-back class, **PacGUI**, is defined which provides a known, well-defined set of such interactions. Such an object must be supplied to each operation each time that operation is invoked.

While standards as to what a particular operation name should connote are being contemplated (for instance, 'Run' will probably connote a batch style run to completion while 'Start' would indicate an interactive operation initiation to be terminated by some later 'Stop'), there is no standard or requirement for the operations that any particular PIA-wrapped application is to provide. Thus it is that the High Speed Research (HSR) Inlet Unstart test support (the first application actually adapted to this architecture) provides neither 'Start' nor 'Stop', but instead provides 'LoadFromFile' and 'CreatePlaceholder' operations.

### 4.1.4 Object Sorting

A fourth structure has been defined and implemented, but it is not shown in the figure. The structure provides a sorting of objects of the application by their derivational heritage. For example, a far-field Mach number parameter object is sorted as being

1. A far-field Mach number parameter,
2. A Mach number parameter,
3. A dimensional, double scalar parameter, which is, in fact, non-dimensional,
4. A dimensional, double scalar parameter,
5. A double, scalar parameter,
6. A scalar parameter,
7. A parameter,
8. A describable application object,
9. A directed graph object,
10. A status-bearing object, and, finally,
11. An object.

The need for this derivationally-exhaustive sorting arises because a particular application may specialize parameters beyond the level that is well known. Continuing the example above, an application may define and use many particular, custom types of far-field Mach number parameters, but have no parameter object instances that are exactly a far-field Mach number parameter as defined for all applications by the architectural standard. Without the derivationally-exhaustive sorting, an inquiring application would have to examine each parameter to determine if it was, in fact, a kind of far-field Mach number, while with the sorting an inquiry can be directed immediately to those specialized

parameter objects even though none may be exactly a far-field Mach number parameter object.

This object-sorting structure has been unused to this point since its utility is principally of use to search engines and the like, which are themselves as yet unrealized.

## 4.2 Operating Context

By now the reader will be impatient to learn the significance of the great, sweeping curves that run from each terminal node of the parameter identification tree, and from each node of the operation identification tree to the lower leftmost node of the configuration tree. The answer is that the objects of both these structures operate within the context of a particular parameter configuration which must be identified when certain functions are invoked.

Identifications and operations offer `IsVisible` and `IsEnabled` member functions, respectively. These operations indicate whether or not their presenting objects are active (after their kind) within the context of the current parameter configuration. By this means, an `Initialize` operation could refuse to work when queried in the context of a parameter configuration that either had or inherited a populated data set. Similarly, an identification of, say, a turbulence model parameter would respond that no such parameter exists if the turbulence model, itself, is turned off in the identified configuration, even though it might well inherit such a parameter from an ancestral configuration in which the model was turned on.

## 5 Configuration Policy

As illuminated above, the application architecture is quite abstract and leaves an enormous amount of room for maneuver in adapting an application to the PIA environment; however, the architecture also leaves considerable latitude to the consuming tool (most commonly, a conforming GUI) to make of things what it will. One of these areas is the configuration policy to be applied when modifying a data item.

As noted in the architecture section, the modification of a data item identifies a new point in the  $n$ -dimensional data space and, customarily, results in the generation of a new configuration object attached to the configuration tree at a point appropriate to inherit all the other unmodified parameter objects. This is a policy which is, itself, not actually provided by the PIA implementation, nor is it necessarily mandated by the architecture. The decision to actually implement this policy is left to the consuming tool.

The reason for leaving this policy decision to the consuming tool is that this is not the only reasonable policy. Several additional policies are suggested in the following subsections and it may be that a GUI might well wish to offer some or all of these policies to the researcher to facilitate the work being conducted.

### 5.1 Replication

This policy is the one initially suggested above. A new configuration is generated and attached, presumably as the direct descendent of the configuration containing the data item to be modified, and the encapsulating parameter object is replicated in that new configuration with the modified value. This might be considered the basic step in design space exploration.

### 5.2 Modification

If the data item is contained in a configuration with no descendent configurations (or, for additional complexity, no descendent configurations inheriting the data item), the encapsulating parameter object can be modified as it resides in the existing configuration, provided that no output results exist in that configuration or that such output results as do exist are either marked invalid or are discarded. This policy, while not so utterly clear as the previous, might be useful when casting about trying to find a meaningful starting point.

### 5.3 Invalidation/Re-execution

In this policy, the data item may be modified despite (indeed, because of) the fact that descendent configurations inherit the data value. Here, the policy goes beyond simply invalidating the output data dependent on the data item to be modified. The policy would specify the automatic re-execution of the application to regenerate those results with the effect that an entire design space might be analyzed as the result of a single act. Note, though, that the semantics of architecturally identifying the re-execution act have not yet been devised; however, if and when the implementation of such a policy should come to hand, the development of appropriate semantics would not seem to be a difficult problem.

### 5.4 Subgraph Replication

The previous policy has the disadvantage of discarding the previous results of the design space represented by the descendents of the configuration. A further extension of the

policy could be to (1) replicate the encapsulating parameter object for modification in a new, sibling configuration, (2) replicate the descendent subgraph inheriting the original value as a descendent subgraph inheriting the modified value and, then, (3) invalidate and regenerate the output results in that replicated subgraph. This policy allows the parametric study of complete design spaces while retaining all of the previously generated results.

## 6 The Base Object

Very nearly all of the object classes involved in implementing the PIA application architecture described above are derived from a common base class, **PacBObj**. This base class provides several key features:

1. The (inherited) ability to participate in a directed graph,
2. The ability to be ‘described’,
3. The ability to transmit declared events, and
4. The ability to traverse upward through the application structure.

### 6.1 Directed Graph Capabilities

The ability to participate in a directed graph allows the direct implementation of the  $n$ -ary trees of the parameter configuration and identification structures through inherited characteristics. An  $n$ -ary tree is, after all, merely a directed, acyclic graph in which only one immediate predecessor is ever allowed.

As will be noted later, further use of the directed graph capability is made.

### 6.2 Descriptive Capabilities

The ability to be described brings a good deal of useful function to the entirety of the architecture. The descriptions that may be added to any such object include but are not limited to

1. A name (with synonyms if desired),
2. A set of access controls,
3. An annotation,
4. A short, descriptive text,

5. A change history,
6. A drop-down prompt,
7. One or more graphical descriptions,
8. A Universal Resource Locator (URL),
9. A descriptive, multi-line text,
10. A type,
11. A measurement unit description, and
12. A related parameter reference.

Indeed, virtually any sort of descriptive element can be devised and added to the repertoire merely by deriving a class from the appropriate descriptive base class (which is, itself, derived from **PacBObj** and may, thus, be described).

Because not all objects will necessarily have descriptive elements and certainly not all objects will have all descriptive elements, it was desired to make the descriptive system one of minimal overhead. Thus, instead of a series of components embedded in the **PacBObj** class, the only fixed component of the descriptive system is a pointer to an organizing head which, if present, ecclasiastically sorts a set of description objects by type. Thus, the only unavoidable overhead is that of a single pointer which, occasionally, may be null.

The only limitation of the descriptive system is that, within a descriptive set, only one instance, at most, of any particular controlling descriptive type (that is, at most one name, one annotation, one URL, etc.) is permitted. This limitation is offset to an extent by the fact that the descriptive system is implemented in a layered, hierarchial manner in parallel with the derivational class hierarchy and that distinct descriptive sets may exist at each such level. Thus, for example, a scalar double kinematic viscosity object might be described at the level of

1. A kinematic viscosity,
2. A viscosity,
3. A dimensional scalar double parameter,
4. A scalar double parameter,
5. A scalar parameter,
6. A parameter, or
7. A **PacBObj** object.

This presumes, of course, that that is the derivational class hierarchy of the object.

Within this descriptive hierarchy, the top-most description (that is, the description of the most derived class level) of any kind is considered to be the preferred description; however, the facilities exist to reveal the entirety of the descriptive hierarchy should it be desired.

### 6.2.1 Engineering Logs

The multi-line, descriptive text form is provided as the possible basis of an engineering log facility. Here an indefinite, expandable number of text string elements in an ordered list can be associated with any **PacBObj** derived object. All that is lacking is for the consuming GUI to provide an appropriate editing facility to access, modify, and update the text. This form does not offer any implicit time stamping; however, if such a feature were desired, it could be added easily enough in a derived class.

### 6.2.2 Change Histories

A change history descriptive form is provided. It provides an implicitly time-stamped, ordered, multi-line descriptive form. This form is implicitly used by the **PacPara** (parameter) base class to record parameter modifications as previous value texts. All currently implemented parameter classes utilize this capability in their corresponding Set-Value services.

### 6.2.3 Access Controls

The final descriptive form worthy of further discussion is the access control description. Again, any **PacBObj**-derived object can attach access control descriptions throughout its descriptive hierarchy. These descriptions, while defined only in terms of base class functionality, typically consist of an ordered list of access control entries against which an identified user is checked. The first entry matching the user provides the access characteristics granted. Should no such match be made, a default policy is applied. Since descriptions, and hence access control descriptions, are themselves derived from **PacBObj**, access controls may be applied to access controls. This recursive loop is broken by a self-controlled access control which provides access control characteristics not only for the object it describes, but for itself as well.

Perhaps the most remarkable thing about the access control descriptive form is that it was added to the descriptive system on the order of a year after the descriptive system had

been designed, implemented and, in a project sense, ‘put to bed’, thus illustrating the power and flexibility of the basic descriptive concepts. It is also important to note that the access control descriptive form (as with all the other descriptive forms) incurs no object ‘cost’ beyond the single map head pointer (which may be null) until it is actually used.

## 6.3 Declared Events

The **PacBObj** base class implements an event facility in which some utilizing entity may attach one or more objects of **PacEvent** derivation to a **PacBObj**-derived object. This event base class defines a number of different event types but, as a base class, provides at most a default response should the event be declared. The **PacBObj** base class implements corresponding event functions which, if invoked, will identify each attached event object and transmit the event declaration to it.

Utilizing code is responsible for developing and attaching derived event class forms which do something meaningful should an event be declared. Just exactly what act occurs is left entirely to the utilizing code. Automated corrections may be applied, email may be sent to a user, a notation made in a log file, or nearly any other thing may occur in response. Further, there is no requirement that the event objects used be of the same kind and, even beyond this, there is no artificial limitation on the number of event objects that may be attached to a particular **PacBObj** object, nor upon the number of **PacBObj** objects that may be attached to a particular event object.

As a facility defined and implemented in the **PacBObj** class, the event mechanism is available throughout virtually all of the application architecture. Applications, parameter configurations, operations, parameters, parameter identifications, descriptions, and more may all declare events for notation, action, or other response by utilizing code.

## 6.4 Upward Reference

The basic direction of the application architecture is down from encompassing components to more specific components. Applications identify operations, configurations, and parameter identifications, configurations identify parameters, parameters identify descriptions, and so on. In implementation, though, it is frequently necessary to traverse in the opposite direction; an operation object may need to identify its application object so that it can then locate the parameter identification structure.

This need is met by a pointer member and supporting code in the **PacBObj** class which references the next higher level element of the application structure. Parameters, for example, reference their containing parameter configuration object.

The supporting code permits traversals of this upward linkage in search of a particular kind of object. Thus, simple requests may traverse several structural levels. Further, such code need not be sensitive to the number of levels skipped to locate the desired structural level. A description seeking to locate the application object need not be concerned whether or not it directly describes a parameter in a configuration of an application; its uplevel reference coding will work equally well if it is a description of a description of a description of a parameter in a configuration of an application, or if it is a description of an identification of an application, or a description of an operation of an application. This allows wide application of coding that must, itself, still traverse the structural form of applications.

## 7 Parameters

Data items to be placed in configurations are encapsulated in objects derived from a common parameter base object, **PacPara**, which is itself derived from **PacBObj**. As mentioned in the previous section, the **PacPara** class implements an implicit change history protocol which notes all the previous values of the encapsulated data item as time-stamped text entries kept in an ordered list.

### 7.1 Dependent Parameters

The parameter base class utilizes the directed graph capabilities inherited by it to implement a dependent parameter mechanism. **PacPara** provides implementing code to regard each successor of a graph in which the presenting object participates as being dependent upon the data value which the presenting object encapsulates. In the event that that value is changed, those objects dependent upon that value (throughout the range of the graph) are informed.

Of somewhat greater complication than this is the realization that the replication of a benefactor parameter requires the replication of its dependent parameters. If a benefactor parameter in a particular configuration is to be replicated and modified in a descendent configuration, the dependent parameters of that original benefactor cannot also be dependents of the replicated parameter. In turn, the replicated (and modified) benefactor cannot simply inherit the dependents of the original parameter since their values (presumably) represent correct dependent values for that original

parameter, not the modified value of the replicated parameter. Thus, **PacPara** must (and does) provide code that will correctly replicate the dependent subgraph of a replicated parameter so that the modified value of the replicated benefactor may be correctly propagated to the replicated dependents in that subgraph.

### 7.2 Infusion of Semantic Meaning into Parameter Objects

The self revelation of kind mechanism provided by the foundation object of the developed class system is used by the parameter object hierarchy to infuse semantic meaning into parameters [3]. The first derivations of parameter objects specialize parameters by their basic structural forms; scalar, vector, matrix, organizational, and the like. Further derivation then associates an atomic kind; long, double, Boolean, string, and the like with these forms as appropriate.

A further specialization of double parameter kinds declares them to be dimensional in nature, that is being a measurement in some system of measurement such as the metric system of measurement. (The concepts of dimensionality are discussed in greater detail in [4].) From this point a great majority of engineering parameters may then be derived, each drawing upon the dimensional base facilities to present themselves in the system of measurement requested.

Many other engineering parameters are non-dimensional. Thus, the next specialization of dimensional objects is, paradoxically, to a non-dimensional form which may then be used as a basis for these non-dimensional parameters. The basing of non-dimensionality upon a dimensional foundation allows the free combination of these parameters with dimensional values. Thus, a Mach number may be multiplied by a computed speed of sound to result in a dimensional speed available in whatever system of units is desired.

The related parameter descriptive mechanism is utilized to associate other parametric information with semantically-defined parameter objects. Consider the following utilization of the capability.

1. A vector of double values is specialized through several layers of derivation to be a one-dimensional-grid of total-pressure values. (The derivation of the class is shown in Figure 7.1.) That kind of parameter object is then defined as associating through the related parameter descriptive mechanism a vector of linear position

measurements which reveal the X-coordinate values of that one-dimensional grid.

2. Another parameter specialization through derivation creates a vector of those one-dimensional-grid total-pressure parameter objects which is declared to be a time-history of those parameters. (The derivation of the class is shown in Figure 7.2.) The related parameter descriptive mechanism is again used to locate a time-value vector the elements of which are defined as being the times associated with the corresponding elements of the time history vector.

By working to the semantic meanings of these two classes, a consumer of the second parameter object may discover it to be a time history of one-dimensional-grid total-pressure values for which it may further obtain (1) the times at which each grid result is valid and (2) the positions of the grid points.

This infusion of semantic meaning through derivational specialization exposed through self revelation of kind and, to an extent, content is an enabling technology for the propagation of information between applications, as will be discussed shortly.

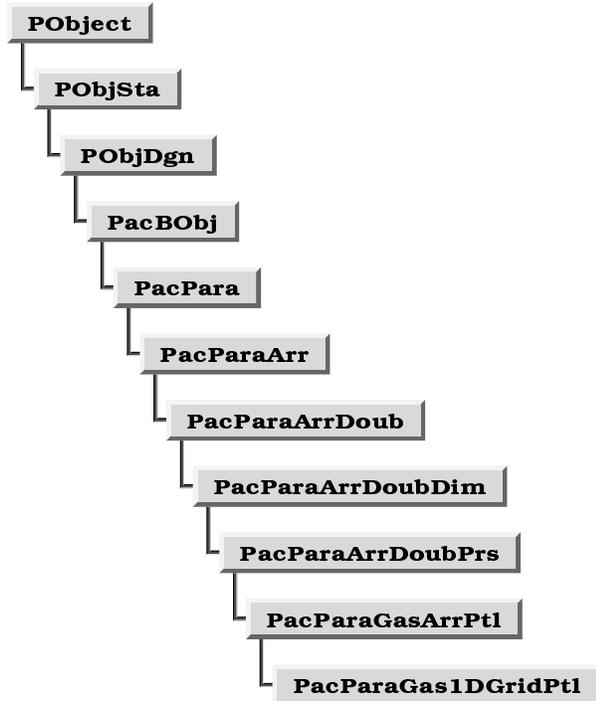


Figure 7.1: Derivation of a One-Dimensional Grid of Total Pressures

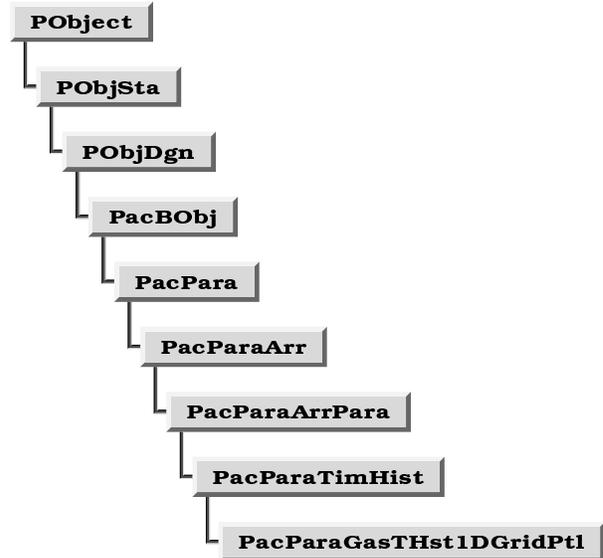


Figure 7.2: Derivation of a Time History of One-Dimensional Grid of Total Pressures

## 8 Persistence

Saving the state of an encapsulated application, whether implicitly or explicitly commanded, is a self-evident requirement and was, in fact, one of the very first challenges confronted. Review of various commercial data base products at the start of the project revealed all of them to be generally intolerant of evolutionary change, which itself was another requirement of the PIA effort. This was not an overriding concern though because, for administrative reasons, these products simply were not available to the project.

To meet the persistence requirement, an object serialization capability was implemented. In this approach object contents are written out to or read back from an archive file (or other repository) under the control of a Serialize function. (The 'Serialize' name is entirely arbitrary.) An archive object keeps track of what objects have been serialized or de-serialized so that redundant references to an object are treated as such and do not cause redundant serializations or de-serializations of the object.

The drawback of this serialization is also its strong point: a Serialize function must be manually coded for every class that might participate in such an operation. For the most part such coding amounts to mere tedium; however, it is in this coding that the groundwork for evolutionary change can be laid. If one takes the precaution of serializing an archive version number as the very first step of object serialization, then conditional code for the de-serialization of

old object versions can be generated when object revisions are defined. By this means, old objects may be recovered even when the base class of a class has been changed. This allows quite extraordinary revisions to an implemented PIA wrapping to be effected without losing old, archived versions of the implementation.

As utilized in the C++ implementation of the architecture, the serialization of objects is explicitly commanded and results in an act which saves the entire application wrapper instance to persistent storage, or recovers that entire object set from storage. As will be discussed later, the Common Object Request Broker Architecture (CORBA) implementation of the architecture is implicitly persistent; however, the same serialization mechanism is, in fact, used, but on an object-by-object basis rather than across the entire application wrapper object set as a whole.

## 9 Information Propagation

The infusion of semantic meaning into parameter objects through derived class specialization and self revelation mechanisms forms the basis for the interapplication transfer of information by allowing one application to ‘look’ at parameters of another application and discern on an automatic basis the semantic nature of the observed parameter objects. This basic technology enables a number of interapplication information transfer modes from user-driven collaborative exchanges through automated browser/search engine harvesting of information to completely automated application graphs for comprehensive engineering analysis of a project as a whole.

Because the propagation of parametric information throughout application graphs could be implemented entirely within the PIA framework, it was the first form of information transfer implemented by the PIA project [5]. The basic goal is as has been previously suggested: to arrange disparate applications into a cooperative graph whose operation carries out all of the analyses relevant to an engineering project as a whole.

Consider as an example of the arrangement of applications into a graph for the purposes of information propagation the situation depicted in Figure 9.1. Here, an analysis control application is made the initial node of the application graph. This pseudo-application exists solely as a convenient point for declaring new configurations of the overall problem and setting parameters within those configurations to control the analysis done. Two read-only applications, a wrapping of CAD geometry information and a wrapping of Particle Imaging Velocimetry data are the initial node’s immediate successors, each leading to the ‘real’ application

in this example, a wrapping of an two-dimensional flow solver. The design of this particular graph, then, is to provide three sources of information to the flow solver, each of those pieces providing an independent part of the input whole for that solver.

The basic conceptual view behind the arrangement of an application graph is that there always exists some source definition of a proposed configuration of the project which then feeds as input to various analyses of that configuration. Those analyses then produce results with two potential aspects: intermediate values which are of use for further forms of analysis and final answers contributing toward a judgement of the engineering merit of the design.

Another aspect implicit in this view of information propagation throughout directed application graphs is that such applications operate in what might be called a batch mode that reliably turns input information into output information. The information propagation scheme as implemented to date does not contemplate an iterative cycle upon the graph until some result balance is achieved. Note, though, that the architecture does not preclude such a formulation at some future point.

Yet another feature of information propagation as presently implemented by the architecture is the forced synchronization of parameter configurations. Information propagation is required by the implementing code to be from a particular parameter configuration and, potentially, the configuration subgraph which it heads, to a precisely corresponding parameter configuration in the receiving application and, potentially, the configuration subgraph which it either heads or which is created for the purposes of propagation with the particular receiving parameter configuration as its header. By this enforced stipulation, the concept of a project configuration as encapsulated by the configuration object scheme is made more real and, it is expected, the problem of mismatched configurations within a project analysis will be eliminated. No longer will the weight of the thin tank wall be combined with the strength of the thick tank wall to produce a winning design in all departments except manufacturing.

To see this synchronization of configurations, consider first the pre-propagation situation depicted for two application graph members in Figure 9.2. The parameter configuration graph of the successor application is, clearly, a subset of the configuration graph of the predecessor application. (The supposition here is that the configuration graph of the successor application, in fact, corresponds exactly to the left portion of the graph of the predecessor application, presumably because of prior acts of information propagation.) After information propagation has occurred, as depicted in

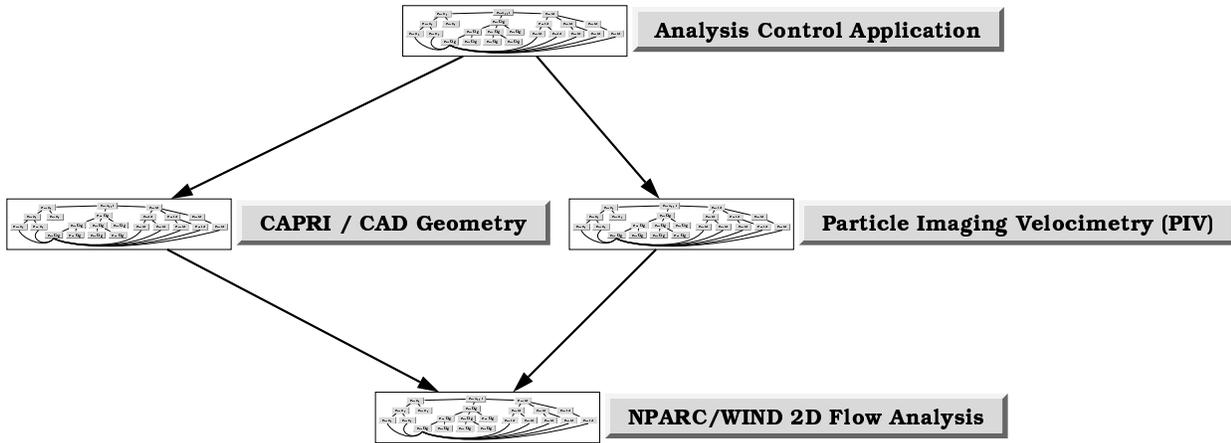


Figure 9.1: A Simple Arrangement of Applications into a Graph for Information Propagation

Figure 9.3, new parameter configuration graph nodes have been created in the graph of the successor application so as to exactly duplicate the configuration node from which the information propagation occurred and to duplicate the subgraph which that configuration node heads, from each element of which information propagation has also occurred.

The information propagation implementation also recognizes that not all applications are entirely reliable in their operation. (Indeed, it was the twitchy nature of sophisticated, high-fidelity CFD codes that gave part of the impetus to the PIA project in the first place.) To deal with this, the information propagation support utilizes the event mechanism built into the **PacBObj** base class to allow inappropriate operations to alert supposedly corrective entities, whether automated or human-interactive. There is, through this facility, the ability to apply corrective measures and re-attempt a particular operation in the overall propagation activity. Failing such corrective actions, the information propagation system will mark the affected parameter configurations as being defective and will prevent further propagative acts based upon those configurations.

The process of information propagation as currently implemented proceeds in the following general manner.

1. The process is begun by delivering a propagation command citing a specific parameter configuration to an application object which is, itself, a member of an application graph. Typically, this application object will be acting as the initial node of that application graph.
2. The application object does what it may to convert its own input information into output information. Should this element of the process fail, information

propagation is discontinued.

3. The application then passes the propagation operation on to each of its immediate successors in the application graph. The identified parameter configuration is passed on in this act.
4. Each receiving successor application establishes that it has a corresponding parameter configuration, or creates such a corresponding parameter configuration in its own configuration graph. Further it verifies that it has, or it creates, a subgraph corresponding to any subgraph headed by the identified source parameter configuration.
5. Each receiving successor application then examines the parameter objects available in the source parameter configuration (and in the subgraph which it may head) and, based upon the semantic meanings revealed by those parameters, acquires such information as it may. The information is encapsulated in the corresponding parameter configuration(s) of the receiving application.

Each receiving successor application is free to examine the extended predecessor applications of its own propagating immediate predecessor application, to the extent that those may exist, to acquire information from the parameters of those applications, too, in the event that not all relevant input is available from its own immediate predecessor applications.

6. When each receiving successor has received a propagation act from each of its own immediate predecessor applications, it then operates so as to convert its own inputs into outputs and then passes the propagation act on to its immediate successors.
7. The propagation of information continues in this manner throughout the graph until terminal nodes of the

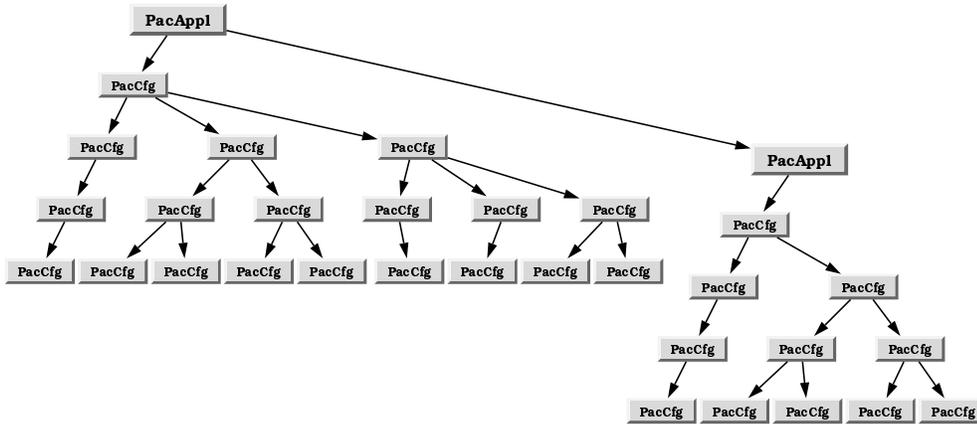


Figure 9.2: Parameter Configuration Graphs of Two Application Graph Members Prior to Information Propagation

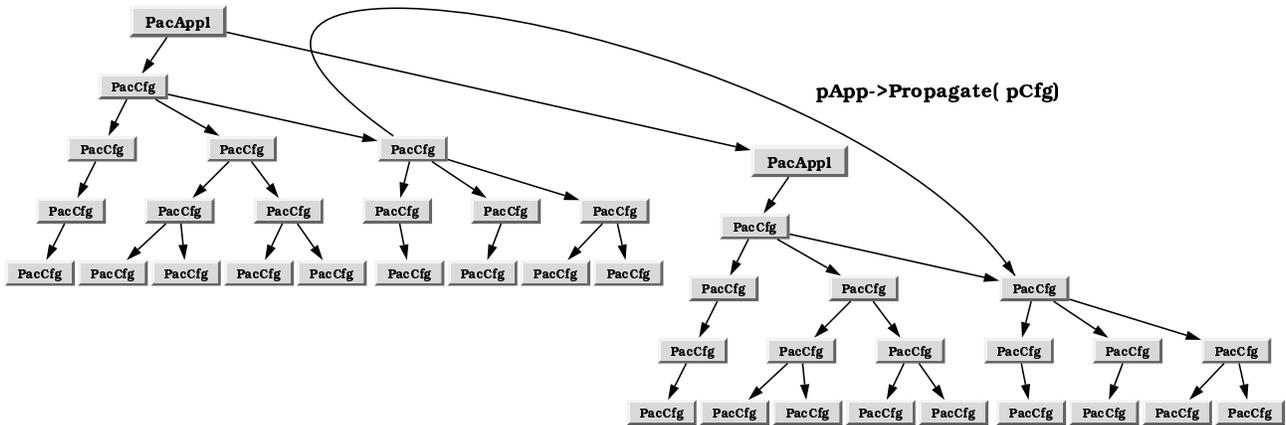


Figure 9.3: Parameter Configuration Graphs of Two Application Graph Members After Information Propagation

graph are reached and, recognizing that they have no successors, those terminal applications return the propagation act back up the graphical chain to the originator of the act.

It should be remembered in all of this that it is the technology of self revelation exposing infused semantic meaning that makes the implementation of information propagation tenable. Applications wrappers need only be coded to look for the kinds of information they desire to acquire during propagation, as in a process of filtering that that is of interest from that that is not. It is not necessary to code for connection to a specific source application to obtain an expected kind of information, nor is it necessary to code for specific topological arrangements of applications.

## 10 Documentation

Complete, class-by-class, member-by-member documentation has been generated in Hyper-Text Markup Language (HTML) format and placed on a central server the Glenn Research Center. The documentation provides not only basic explanatory text as to what particular components do, but also tries, when appropriate, to discuss why particular choices were made, what expectations exist for the use of particular capabilities, and the like. The root URL for this documentation is

<http://www.lerc.nasa.gov/WWW/price000/index.html>

It must be strongly emphasized that these pages are the informal generation of the researcher involved and do not, in any way, shape, or form, represent an official statement of the Government of the United States.

## 11 Experience

To date, three applications have been wrapped in the C++ implementation of the PIA Application Architecture: a presentation of experimental data from an inlet unstart experiment for the High Speed Research project (known as HIU), a presentation of flowpath geometry information from Computer Aided Design (CAD) sources accessed through the Computational Analysis Programming Interface (CAPRI) cross-vendor package, and an operational wrapping of the Large Perturbation Inlet (LAPIN) analysis code.

### 11.1 HSR Inlet Unstart

The wrapping of the HIU test was encouraging because, up until the invitation to do that effort, experimental aspects of the PIA task had been greatly de-emphasized. Penetration of the ICE project into the experimental arena had been shallow and, as a consequence, experimental data handling by the PIA project was considered of minor importance. Thus, it was gratifying to see the concepts conceived almost entirely with analytical tools in mind bent so amiably to the needs of the experimental environment.

On the other hand, the experience with the HIU test has instantly and forcibly demonstrated the inadequacy of the single virtual address space in which the C++ implementation of the PIA effort must live. The HIU test has generated something on the order of 25 GigaBytes of experimental data. Without lapsing to a meta-data concept, such a data load is utterly crushing to the C++ implementation environment. While a meta-data solution is possible, other growth directions (to be discussed shortly) mitigate against such an implementation, even as a short-term solution.

One judgement that was derived from the HIU experience is that the 25 GigaByte size was not an unusual thing. Experimental propulsion efforts are expected to often have data volumes of this magnitude. Further, a comprehensive CFD investigation saving all intermediate results and steps as envisioned by the architecture could also produce data volumes of this magnitude. Given the fact that the PIA effort is to allow the researcher to browse from one such volume of data to the next and, ultimately, migrate information content between such repositories, the need to expand the implementation into a more accommodating form is unavoidably clear.

It should be noted that the HIU implementation predated all of the actual work related to the infusion of semantic meaning into parameter objects. This was an expedient guided by scheduling factors. Because no information propaga-

tion activity was projected within the useful lifetime of the HIU application (indeed, information propagation was only a distantly conceived notion at that time), this expedient was considered entirely acceptable.

#### 11.1.1 HIU Implementation

The data to be managed in the HIU application consists of time-series data streams sampled at several thousand samples per second from each of some 150 different data sources. For the most part these data sources consist of high-response pressure transducers variously distributed between static and total pressure measurements; however, a number of other inlet, engine, and free stream data sources are also involved. The data from an individual data point is provided as two computer files: an interleaved, binary data file of all the data sources in round-robin order and an associated, text-based format file identifying the precise contents of the binary file.

It was immediately determined that each individual data point from the experiment would be loaded into a single, private parameter configuration object. To allow the researcher to structure configuration objects into an easily comprehensible arrangement, it was also decided to provide a placeholder capability by which empty configuration objects could exist as the direct parent of a series of related data points.

Because of the essentially static nature of experimental data, the HIU application required little further functionality. The following tasks were the only ones of any significance.

1. An operation to convert an empty **PacCfg** object into a placeholder was developed. In the conversion process, the subject **PacCfg** object took on a non-empty appearance.
2. Several operations were developed to read a coordinated format and data file pair and place the parameter-encapsulated information into an empty **PacCfg** object.
3. A **CreateApplication** member function (of the **HiuAppl** application class derived from the **PacAppl** class) was developed. The main task of this member function override was to construct the identification structure identifying all the data elements of a test point.

#### The Placeholder Operation

The placeholder operation is quite simple. A protocol internal to the HIU application wrapper was established by which the presence of a reading number parameter, known internally by the name 'RdngNo', was considered to make a configuration non-empty. Since the data loading operations would be coded to refuse to operate if a reading number parameter were present in the target configuration, it was only necessary for the placeholder operation to place an empty reading number parameter in the selected configuration to give it a non-empty, placeholder appearance.

As a side effect of the operation, the user is prompted to provide a name for the placeholder configuration which is then set into the descriptive system for the target **PacCfg** object. This allows the placeholder object to exhibit a useful, memory-jogging name in all further operations.

### The Data Loading Operations

Three data loading operation classes are, in fact, currently provided by the HIU application. Each differs only in the expected layout of an individual binary value in the data file. This difference is effected by a function override in data loading operation classes derived from the base data loading operation. Thus, the real guts of the data loading operation was written only once and was inherited by the variant forms.

As would be deduced from the description of the placeholder operation, the first significant step of data loading is to assure that the target configuration object is, in fact, empty based upon the established internal reading number parameter protocol. With that matter satisfactorily resolved, the user is then prompted via a supplied **PacGUI** object to provide the names of the format and data files.

Most of the data loading operation is relatively uninspiring; however, a few points of interest do exist which illustrate the sort of implementation freedom which exists behind the PIA architectural wall.

The first of these interesting points is the handling of the mismatch between the data item names used in the format file and the corresponding names used in the HIU application. The experimentalists, out of whatever basis seemed reasonable to them, used strictly upper-case alphanumeric identifiers. The HIU application elected to use mixed-case identifiers to achieve a more esthetically pleasing appearance. To bridge the gap between these two selections, a synonym table was added to the developed application class, **HiuAppl**. When the format file specifies a data item name that is not directly found in the identification structure, an attempt is made to resolve the problem through

application of the synonym table. Since this protocol is entirely internal to the HIU application, the addition of the facility to the derived application class is entirely appropriate.

A related extension was touched on above: the location of data items by name in the identification structure. It was considered desirable not to code the data loading operation with explicit knowledge of the structuralization of the data encapsulated by the  $n$ -ary tree of the application data identification structure. (To do otherwise would mean that each alteration of the identification tree would have to be matched by a coding change to the data loading operations.) On the other hand, the  $n$ -ary tree of the identification structure offers no find-by-name facility since there is no requirement that nodes of the tree have names unique across the tree, even though that is in fact the case in this particular application. (The directed graph class upon which the identification structure is based does have a find capacity, but it consumes an ordered set of names to make branch selections as each node of the tree is traversed.) Thus, it was considered expedient, though not absolutely necessary, to provide in the patriarch of the identification graph a map sorting identification objects by their simple names. By consulting this map, the data loading operation is able to locate the identification object for a particular data item, for instance the 'RdngNo' parameter, directly. Again, since this mapping is a matter entirely internal to the HIU application, it is something appropriately done behind the PIA architectural wall.

The data loading operation provides a non-catastrophic response in the event that the format file identifies a data item that cannot be located in the existing identification structure. Subject to the consent of the user through the provided **PacGUI** interaction object, an entry is made in the 'Found' category of the identification structure using the name supplied by the format file for the data item. Thus, even though the wrapping of the HIU experiment is a relatively static thing involving the production of programming language code, a certain modest flexibility to adapt to new data items on the fly is built in.

With regard to the actual mechanics of data loading, the processing of the format file identifies the order of the data items contained in the binary data file. As each such item is identified, its identification object is located (through the various internal mechanisms just discussed) and used to obtain the fully-qualified name by which the corresponding parameter object is to be sorted within the parameter configuration. A new parameter object is obtained (in point of fact, a **PacParaArrDoub** object), the fully-qualified name associated with it, and it is placed in the data configuration object. A pointer to the parameter object is also recorded

in an ordered list called the interleave list and automatic change notation is turned off in the parameter object for the duration of data loading. When all of this is accomplished and the format file is exhausted, data loading commences (after appropriate initialization of indices to 0) by successively obtaining the next binary item from the binary file and placing it in the next slot of the next parameter object in the interleave list. As the process proceeds, the interleave list is indexed most rapidly. Each time the interleave list end is reached its index is reset to 0 and the next item counter for the parameter arrays is incremented. When the binary file is exhausted, data loading stops.

As mentioned previously, the actual obtaining of a single binary data item is encapsulated in a separate member function of the data loading operation object. The base class implementation of the data loading operation simply assumes that binary data items are double items (that is, the primitive C++ data type **double**) in the native form of the executing machine. The other two derived operation classes differ from the base class only in overriding the data item acquisition function to load items as **float** items in native format or as **float** items in byte-reversed format.

## Identification Extension

An adjustment to the data identification process was needed in the HIU application. As touched on above in the description of the data loading operation, the need to locate identification objects directly by the simple name of the data item was met. This capacity was introduced in the **HiuPid** class which was directly derived from the **PacPid** class. The actual creation of the empty map was added to an override of the `CreateInitialNode` member function and additions to the map were made automatic through an override of the `AddSuccessor` member function. Since an object of the **HiuPid** class is used as the patriarch of the identification graph and since the inherited `AddMember` function assures that a graph may only add members that are of or derived from the class of the patriarch, it is certain that every identification added to the HIU identification graph will necessarily make an entry in this internally-defined map.

## CreateApplication Function

The implementation of the `CreateApplication` member function, while being a great mass of tedious drudgery, is not particularly remarkable given the discussion above. A few remarks are more than enough.

A script engine is executed to build the identification graph, organizing the potpourri of some 150 data items provided

by the experimentalists into more manageable groupings thought to be appropriate to the situation. By using identification objects of the **HiuPid** class, the internal mapping from simple name to associated identification object is automatically constructed.

In building identifications, it was recognized that in an experimental data situation, the parameter inheritance mechanism of the parameter configuration graph should be turned off. In the course of testing, data items are lost due to instrumentation failures and the like. Sometimes bad data are simply recorded and a record made that they are, in fact, bad. Other systems succeed (often at some point down stream from the point of acquisition) in discarding such bad items. It was realized that should such bad items be discarded by the time of presentation through the HIU PIA wrapper, values inherited from previous configurations when the item was good would not be appropriate for display. Thus, a small feature of the identification class requiring that a parameter actually exist in the identified configuration was turned on to cure this problem. In practice, the act required far less effort than its explanation.

A second script engine builds the synonym table in the **HiuAppl** application object for the known text differences between the experimental nomenclature and the names used by the HIU application. Yet a third such engine obtains instances of each of the implemented operations objects and adds them to the application.

### 11.1.2 HIU Work Not Completed

The HIU implementation reported above was considered as only an initial plunge into the demonstration. The experimentalist involved wanted not merely to see and browse the data, but to reduce that data through a number of defined computations resulting in derived quantities such as the local Mach number. It was planned to implement such calculations as additional operations of the HIU application, in accordance with general PIA design; however, due to the demise of the entire High Speed Research Project, it is doubtful that this work will ever be done, at least for the HIU application.

This missing effort is brought up to illustrate a philosophical point of PIA design: that, to the extent the HIU application is different from every other application, such computations are appropriate specializations of the application that are well encapsulated behind the **PacOp** operation object architectural wall. A point of discussion may arise, though, should one care to assert that the computation of things such as local Mach number is, in fact, something common across many applications. To the extent such an

assertion might be true, such operations would then properly fall beyond the PIA architectural wall into the province of the code consuming the application, that being generally a GUI. Since the PIA architecture does not constrain itself to any particular operating environment, it might be that different, specialized GUIs could be developed to address such common, repeated operational needs.

Additionally, it should be noted that, even though a particular PIA application might provide such specialized operations, the architecture cannot prevent consuming environments from applying such inferential computations to data which is exposed. This application of consuming environment operations becomes more likely, and is indeed enabled, as more semantic information is provided by the derivational specialization of parameter objects. In the HIU application all data was placed in **PacParaArrDoub** parameter objects for the simple reason that PIA work had not yet developed any more semantically meaningful classes. As classes identifying the data as something on the order of fluid flow total pressure, fluid flow static pressure, and the like, become available, it is likely that specific, specialized consuming environments will be able and inclined to provide common, computationally derived quantities even though the specific application might also provide such quantities.

Also, it is expected that many parameter forms will provide widely-applicable functionality peculiar to their kind. For example, conversions from experimental values obtained in differing flow regimes (for example, subsonic, transsonic, supersonic, and hypersonic) to 'true' total pressure values would be well included in a total pressure parameter class where they could be utilized by experimental application wrappers.

## 11.2 Cross-Vendor CAD Access

A wrapping application was developed which presents geometry information developed from CAD information obtained through the cross-vendor CAPRI application programming interface. This wrapping is reported in detail in a companion publication [6].

The key achievement of this wrapper was to present a **PacParaGeoBdry** boundary object, along with its supporting component objects. This object demonstrates key elements of the parameter object concept. First, by its revelation of kind, it presents the semantics of a logically-complete boundary built upon the concatenation of a number of open geometric faces. Then, after being recognized as a boundary object, a number of defined services, among these the ability to obtain open and closed cross sections and to com-

pute the area of such closed cross sections as might be obtained. As will be discussed shortly, the Large Perturbation Inlet (LAPIN) analysis code used this object with its known services to obtain needed geometric input information.

Another aspect of the geometry wrapper was the considerable behind-the-scenes maneuvering that went on in achieving its function. The implementation of the CAPRI interface and the CAD toolkit underneath that interface precluded a straightforward parent to child to grandchild program structure to obtain the desired geometric information. Instead, it was necessary for the wrapping process layer to invoke a shell script (in actuality, a DOS batch file), which in turn invoked the CAD toolkit, which at the direction of the shell script connected to a Dynamic Link Library (DLL) created to implement the needed geometry extraction functionality, which in turn created the geometric parameter objects and 'piped' them through to the patriarchal wrapper application layer by serializing them to a file from which that patriarchal layer subsequently deserialized them. All in all, a rather convoluted approach to information retrieval, and still entirely transparent to the consuming code (in this case, the PIA testbed GUI). As did the HIU experience in a different way, this demonstrates the nature of the architectural mechanism: provided functionality (the acquisition of geometry data) was obtained through an apparently straightforward interface while, in fact, highly convoluted maneuvers occurred to implement that interface.

## 11.3 Large Perturbation Inlet

As with the geometry wrapper, the wrapper to the LAPIN code is reported fully elsewhere [7]. This wrapper is more representative of the expectations for a typical application wrapper. It presents the comprehensive data set of the LAPIN application and several operations. These operations include the ability to load the parameter set from sources traditional for the legacy application and to operate the code and recover its output.

Of particular interest here is the parts of the wrapper supporting information propagation. The specializations provided by this wrapper are, as expected by the general design of information propagation, relatively narrow, consisting principally of code for the parameter harvesting stage of the information propagation act. The implementation is in the form of a filter looking for parameter objects of the kind **PacParaGeoBdry**, the very kind the geometry wrapper works to present.

When the parameter harvesting operation is complete, the following decision tree is executed.

1. A geometric item is selected. A geometric assembly is selected in preference to geometric boundaries. If other than exactly one geometric item parameter has been identified, the geometry part of the information propagation process is abandoned.

Currently, there is no discrimination applied between multiple geomtric items; however, mechanisms do exist by which inappropriate items might be excluded from consideration so that exactly one geometric parameter survives the harvesting process.

2. The geometric item is sectioned in the (X:Y) plane, which is a service provided by both the **PacParaGeoAsmb** and **PacParaGeoBdry** classes.
3. Heuristics are applied to the obtained cross-section curves to identify two open sections taken to be those of the flow path. If two open sections cannot be identified by these rules, the geometry part of the information propagation process is abandoned.
4. The obtained open sections are sorted and ordered to proceed radially outward (that is, in ascending arithmetic order for the Y coordinate values) and from fore to aft (that is, in ascending arithmetic order for the X coordinate values).
5. If the two sections are mirror images of each other (a service provided by the **PacGeoCurv** class which encapsulates sectioning curves), a LAPIN type 0 inlet formulation is generated from the outer curve and the geometry part of information propagation is concluded.

The type 0 inlet designation is an internal formulation of LAPIN and merely designates an axisymmetric inlet with no centerbody.

6. If the first section curve begins on the X axis (that is, if the first curve point has a Y coordinate value that is approximately 0.0), a LAPIN type 1 inlet formulation with an axisymmetric assumption is generated from the two section curves.

The type 1 inlet is, again, a designation internal to the LAPIN code indicating an inlet with a translating centerbody.

7. Should the decision process reach this point, the only option (currently) left is that of a LAPIN type 1 inlet with a two-dimensional (the alternative to axisymmetric) assumption. Cowl and centerbody geometry is generated from the two section curves. Duct width geometry is computed to result in cross-sectional areas matching those obtained from the geometric item parameter object. (The computation of cross-sectional area is another service of the **PacParaGeoAsmb** and **PacParaGeoBdry** classes.)

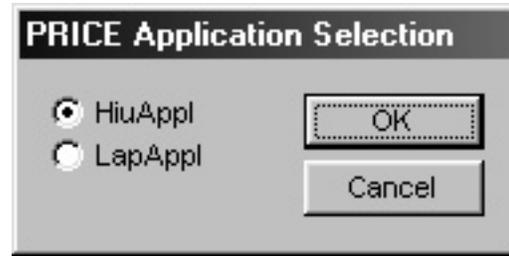


Figure 11.1: GUI Opening Application Query

The interplay of parameter with consumer is illustrated in the above process. The sectioning of boundaries and the computation of cross-sectional areas are both tasks that are considered to be relevant and common across a wide variety of consumers of objects of the **PacParaGeoAsmb** and **PacParaGeoBdry** classes. Thus, once the LAPIN propagation code identifies an object of that class, it is assured not only of the kind of information available, but of the services available for the useful transformation of that information.

#### 11.4 Testbed GUI

Although a GUI is not considered to be a product of the overall project, such a tool is nevertheless necessary for testing purposes. Indeed, a GUI is the most expedient way to see that the concepts described above do, in fact, work.

The first demonstration of the architecture is shown in Figure 11.1. This is a screen capture of the application selection dialog box implemented by the GUI. The dialog allows the user to select one of the available application types through a mutually-exclusive radio button interaction.

The remarkable thing about the application selection dialog is that it is generated on-the-fly by the GUI, rather than by a static coding of the dialog. At the time of dialog initialization, a scan is done of all PIA classes, isolating those that are derived from the type **PacAppl**. (The class **PacAppl** itself is excluded from this set.) A radio button is generated for each such identified application class, drawing the name text from the supporting class information. Thus, the figure shows that, at the time this dialog was captured, two applications were supported: the HSR Inlet Unstart (**HiuAppl**) application and the LAPIN (**LapAppl**) application.

In all of the GUI, there is only one spot in which application-specific coding exists: in the implementation of the document class a series of include statements transmit comments to the linker that cause it to incorporate the class code of the various PIA library components, even though there is no reference to those classes and, thus, no need ap-

parent to the linker for that supporting code. It is expected that once the migration of the architecture to the distributed object environment of CORBA is complete, the need to forcibly include code beyond the generic, well-known library levels will cease to exist, allowing new applications to be introduced to the system without the necessity of re-compiling every consuming tool.

Figure 11.2 illustrates nearly all the rest of the features of the architecture as exercised by the testbed GUI. The window labeled **PacAppl1:1** views a LAPIN application chosen from an application selection dialog in the course of GUI startup. The window labeled **PacAppl1:2** views a second application, in this case an HIU application, that has been created as a successor to the LAPIN application in the application graph.

A window viewing an application lists from top to bottom

1. The parameter identification tree (which is not expanded in either window of the figure for reasons of space),
2. The parameter configuration graph,
3. The operation tree,
4. The application graph predecessor list (except in the case of the **PacAppl1:1** window which views the initial node of the graph which, by definition, has no predecessors), and
5. The application graph successor list.

Comparing the differences between the two windows, in particular the different operations lists, shows the self-revealing nature of applications within the architecture. The **PacAppl1:1** window lists the various LAPIN operations; the creation of LAPIN parameters, the loading of LAPIN parameters from traditional Fortran namelist input, the running of LAPIN, and the validation of the parameter set as input to a potential LAPIN run. Meanwhile, the **PacAppl1:2** HIU window lists an entirely different set of operations; the conversion of a parameter configuration to a place holder and several different data-loading operations, all as previously discussed. These differences are all as a direct result of the GUI inquiring of the application as to the application's content and generating an appropriate viewing window in response.

The two viewing windows also reveal the connection between the two applications as participants in an application graph. The **PacAppl1:1** window views the initial node of

the application graph, as witnessed by the absence of a Predecessor Applications element in its view. (An application is made an initial node of an application graph as an implicit part of the OnNewDocument process.) The Successor Applications list of that view shows the HIU application viewed by the **PacAppl1:2** window as its successor. The HIU application, in turn, lists the LAPIN application of the **PacAppl1:1** window as its predecessor. The application graph can be, of course, expanded to the practical limits of the host machine; it is only for reasons of space in the figure that just two applications in a direct parent child relationship are shown.

The operation of the information propagation process throughout the application graph is also illustrated by Figure 11.2. The parameter configuration graph of the LAPIN application viewed through the **PacAppl1:1** window has been expanded beyond its default single-patriarch form to include two child configurations and a grandchild configuration attached to the second child. By the act of information propagation citing the root parameter configuration of the LAPIN application (effected by first selecting the root parameter configuration of that application and then double-clicking the application element of the viewing window), that parameter configuration graph is replicated in the successor HIU application. This is further confirmed by the fact that the default parameter configuration object names generated in the LAPIN application as the configuration graph was expanded (for example, LapCfg:00F939D0) are, in fact, replicated in the configuration graph of the HIU application. This is precisely as expected by the act of information propagation as a result of its effort to keep parameter configuration synchronized between cooperating applications.

## 12 Future Directions

At this point, the road ahead for the PIA project seems relatively clear. The key technology of self-revelation and its ability to enable common tools, information propagation throughout an application graph, and the like can be considered well demonstrated. Further work now must center on two areas: making the application architecture practicable by moving it to a distributed object environment, and filling in the many semantic gaps so as to have a fully populated set of information forms.

### 12.1 Distributed Object Implementation

As noted in the discussion of the HIU application, the demands of real applications easily overwhelm the capacities of a single virtual address space implementation of the ap-

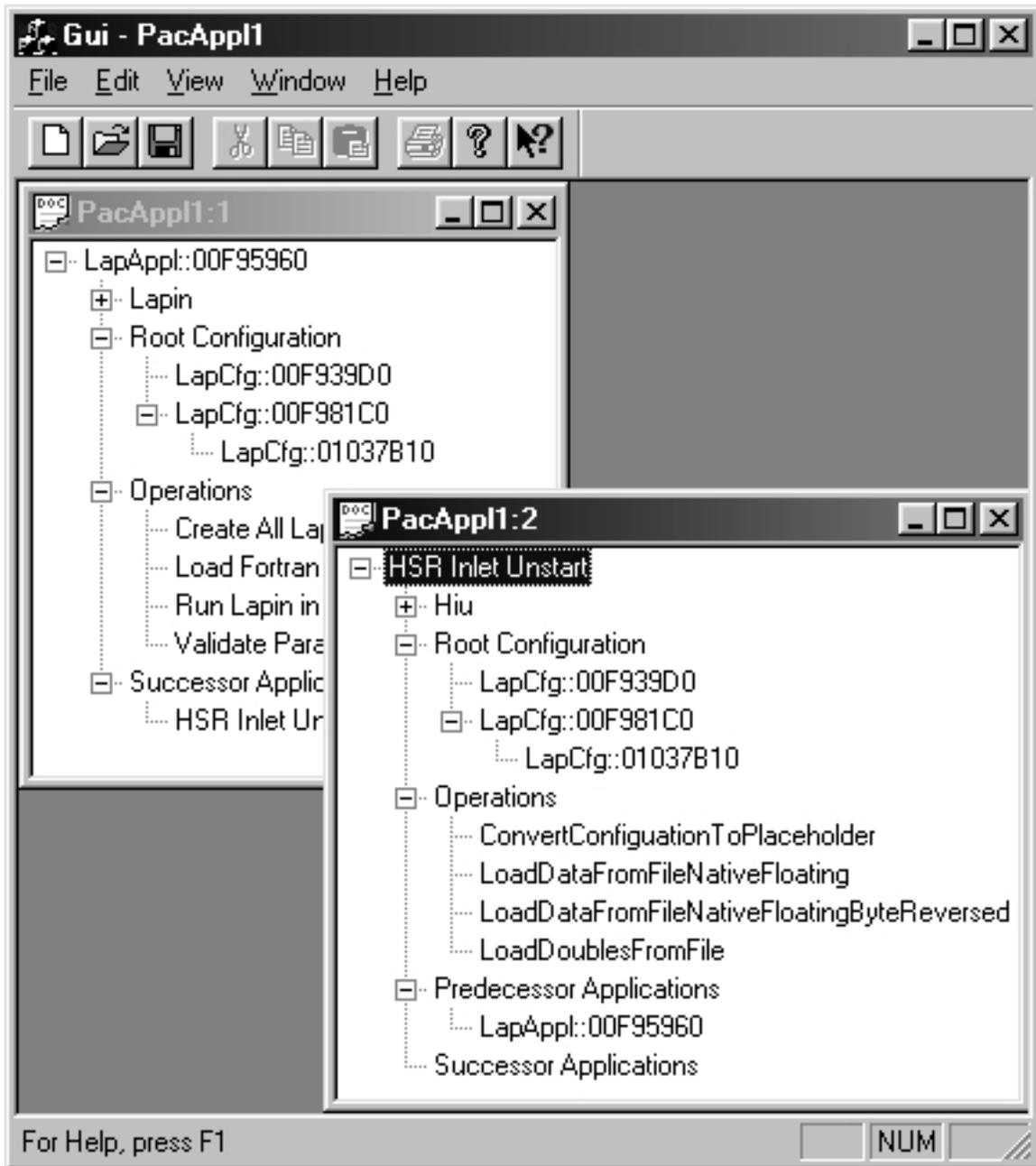


Figure 11.2: Display of Two Independent Applications Connected in a Graph

plication architecture. There is no possibility of accommodating multiple applications in a cooperative graph when even a single application is beyond the range of reason.

For this reason of practicality alone, it is necessary to migrate the application architecture to a distributed, served-object architecture. This work has already begun utilizing the Common Object Request Broker Architecture (CORBA) technology standardized by the Object Management Group (OMG, <http://www.omg.org>).

### 12.1.1 Distributed Object Fundamentals

The basic idea of the distributed object is to convince client code that it is using an object within its own operational space while, in fact, the actual object exists somewhere remote from the client. Frequently, the idea of remote means that the object exists on another machine accessed across a network, not uncommonly that network being the Internet.

To understand the underlying mechanisms of distributed objects as implemented by the CORBA standard, consider the conceptual diagram presented in Figure 12.1. Consuming code at the upper left has pointers or references to what are, in fact, client stubs of the distributed objects it believes it is working with. When a method is invoked on a client stub, instead of performing the requested operation, the method invocation along with its arguments are passed through to the Object Request Broker (ORB) resident in the consuming code which marshals the information into a transportable message. The message is then routed, frequently over the Internet, to a serving ORB. That ORB cooperates with an object adapter (in earlier formulations a Basic Object Adapter or BOA and, more recently, a Portable Object Adapter or POA) to demarshal the message and transmit the method invocation to an implementation skeleton object, which in many cases, simply passes the invocation on to a final, implementing object.

When the remote method operation is complete, the process is simply reversed. The results of the operation are passed back through the skeleton object to the [BOA/POA]/ORB combination, which marshals them into a responding message and routes that message back to the ORB serving the client. The response is demarshaled and passed back through the client stub objects to the waiting consumer code. Except for the generally longer response times, the consuming code is unaware that the operation did not occur within its locally held object.

### 12.1.2 Distributed Object Advantages

This distributed object technology brings with it a number of key advances.

1. The CORBA standard offers a particularly relevant feature called object service activation. Simply put, a served object need not actively exist at all times, but may instead reside in a dormant (or **etherealized** in CORBA terminology) state on some secondary storage device. Should a method invocation come in for such an object, a protocol exists which allows the server to first re-create (or **incarnate**) that object from its dormant form before method delivery occurs. At some later time, should the served object become inactive, it may be placed back in its dormant state.

The key contribution here is that not all the objects of a given application need to be active within the address space of a server at any given time. An application such as HIU may well have terabytes of data, but only that data actively in use at any given moment needs to be served by such a server.

2. Distributed object technology allows PIA applications to be served to consuming tools (and other applications) by multiple serving machines. Thus, it is not necessary to get all of the data and implementing code of all the wrapped applications onto one physical machine.

This ability to serve different wrapped applications from different machines allows those machines to be located with the groups supporting the applications. For example, experimental data applications can be served from machines supported by the data acquisition group while a consuming analysis application can be served by a machine provided by the group generating and supporting that application. Furthermore, both such applications can be used by a consuming client widely separated from both distributed object servers.

3. Distributed object technology separates the issues of functionality from implementation. Thus, the services of a PIA-wrapped application may be supplied through distributed objects without exposing the implementation of those services.

This feature is of particular interest to commercial providers of applications who may wish to sell the services of a particular application without revealing the proprietary methods by which those services are achieved. The PIA technology further facilitates this by allowing general-purpose tools of potentially wide availability to interact with such provided services, thus eliminating the need for a custom access toolkit for every such offered product.

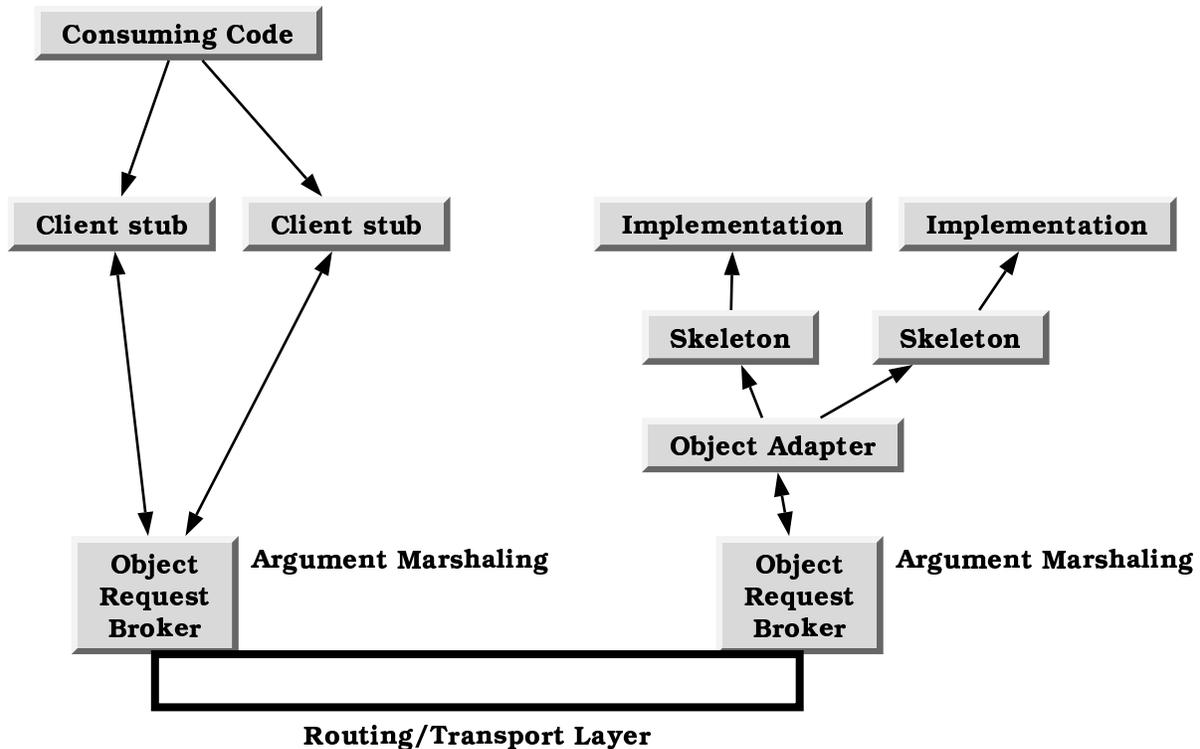


Figure 12.1: CORBA Distributed Object Technology

### 12.1.3 Distributed Object Difficulties

The migration to a distributed object environment, while necessary and advantageous overall, does bring with it certain difficulties.

1. Distributed objects, being exposed upon the net by well-known services, are accessible by, literally, the entire community of the net. Mechanisms must be implemented to limit the accessibility of distributed objects to those that should have access.
2. Having presumably eliminated those who should not access distributed objects of an PIA implementation, it must still be recognized that the remaining accessor set has more than one element. It is, thus, still necessary to arrange mechanisms to assure the integrity of PIA structural components even when quasi-simultaneously accessed by members of that consuming set [8].

### 12.1.4 Distributed Object Persistence

In the CORBA standard, the connection between client and object does not define the period of existence of the object. An object may exist prior to a client connecting to it and it

may exist after the client has fulfilled its desires and exited. Indeed, the CORBA standard provides no specification for either the beginning or ending of a served object. The standard leaves such issues to the discretion of the application.

The PIA project provides answers for the question of the beginning and ending of objects. The class (or, in the CORBA nomenclature, the interface) information support system provides a `CreateInterface` method which creates a new instance of the supported class. To end such a created object, every class implemented by the PIA project inherits from the base class a `SetDefunct` method which declares the termination of the presenting instance.

Between beginning and ending, the CORBA objects of the PIA implementation exist without regard to the comings and goings of clients. As indicated previously, when such an object appears to be idle, the server program arranges to remove the object from active service and saves its content to persistent storage through the serialization mechanisms originated in the C++ implementation work. When a method invocation on such an object arrives at some later time, the object is re-created and its contents restored by a deserialization operation before method delivery proceeds.

## 12.2 Populating the Semantic Set

The mechanism of self-revelation of kind depends upon exacting definition of the semantic nature of a particular object. In the work to date this has been relatively easy since the definitions involved structural components; applications in a generic sense, parameter identifications, configurations, and the like. Even the geometric parameter kinds were defined with comparative ease since the issue of geometry is relatively well settled.

The usability of the PIA technology is closely related to the supply of 'building blocks' available to the application wrapper, particularly to the supply of semantically defined parameter classes from which to choose. While the coding of parameters is often quite trivial, the need for experts in the various disciplines who can formulate clear, broadly applicable definitions of particular parameter forms cannot be overemphasized.

## 13 Summary

An abstract, highly flexible, object-oriented application architecture has been defined. The architecture has been implemented in C++ and real applications have been wrapped according to that architecture. Applications wrapped in this manner have been connected into directed application graphs and the automatic propagation of information from source application to consumer has been demonstrated.

## References

- [1] James Douglas Stegeman, Richard A. Blech, Theresa Louise Benyo, and William Henry Jones. Integrated CFD and Experiments (ICE): Project Summary. Technical memorandum NASA/TM-2001-210610, National Aeronautics and Space Administration, Lewis Research Center, 21000 Brookpark Road, Cleveland, OH 44135, December 2001.
- [2] The American Society of Mechanical Engineers. *Integrated CFD and Experiments Real-Time Data Acquisition Development*, number ASME 93-GT-97, 345 E. 47th St., New York, N.Y. 10017, May 1993. Presented at the International Gas Turbine and Aeroengine Congress and Exposition; Cincinnati, Ohio.
- [3] William Henry Jones. Project Integration Architecture: Formulation of Semantic Parameters. Draft paper available on central PIA web site, January 2000.
- [4] William Henry Jones. Project Integration Architecture: Formulation of Dimensionality in Semantic Parameters. Draft paper available on central PIA web site, March 2000.
- [5] William Henry Jones. Project Integration Architecture: Inter-Application Propagation of Information. Draft paper available on central PIA web site, December 1999.
- [6] American Institute of Aeronautics and Astronautics. *Project Integration Architecture (PIA) and Computational Analysis Programming Interface (CAPRI) for Accessing Geometry Data from CAD Files*, number 2002-0750. Aerospace Sciences Meeting and Exhibit, Reno, NV.
- [7] William Henry Jones. Project Integration Architecture: Wrapping of the Large Perturbation Inlet (LAPIN) Analysis Code. Draft paper available on central PIA web site, March 2001.
- [8] William Henry Jones. Project Integration Architecture: Distributed Lock Management, Deadlock Detection, and Set Iteration. Draft paper available on central PIA web site, April 1999.