INTERFACES AND THEIR MANAGEMENT IN A LARGE ADA PROJECT

Walter E. Zink, Senior Systems Engineer
and
Richard E. Poupore, Systems Engineer
CAE-Link Corporation, Binghamton, NY

ABSTRACT

The Department of Defense continues to require that Ada be the sole programming language for all new software related projects. In addition, these new projects are expected to achieve higher levels of maintainability from a software perspective. Ada and its related compilation/software engineering issues have given interfaces and their management a whole new perspective. In today's environment of dwindling defense dollars, extensive rework during the development or maintenance phase of a project due to interface changes, is prohibitive. Therefore, it is crucial to the success of large Ada projects to address interface issues from the highest perspective. For example, in a simulation environment, as the real-world device changes, the simulator must remain concurrent to provide maximum training benefit. These changes often result in changes to interfaces. In order to keep pace with the development and subsequent upgrades, it is necessary to provide reliable, maintainable and flexible interface structures. By combining a successful software architecture, a database-driven interface management tool and auto-generated connection software, major interface updates can be made in a timely and efficient manner. Experience has shown that with the proper interface design strategy, maximum cost savings can be realized over the entire life cycle of the simulator. An approach to interfaces, their management and connection software is discussed.

ABOUT THE AUTHORS

Walter E. Zink Sr. is currently a Senior System Engineer at CAE-Link Corporation, where he has held management and technical positions since 1986. His current effort is devoted primarily to system-level software design of the Real-Time Simulation Environment that supports the Ada architecture in use at Link. Prior to that, Mr. Zink was employed by the General Electric Company for twelve years, where he held various technical and management positions. His principal activities there were in the areas of Computer Based Instruction and Artificial Intelligence/Expert Systems. Mr. Zink holds a B. S. in physics from Harding University. Other publications includes a paper entitled "The Challenges of Developing A Real-Time Environment in Ada", presented at the 13th Interservice/Industry Training Systems Conference (1991).

Richard E. Poupore is a Systems Engineer at CAE-Link Corporation, where he has been employed since 1987. He is currently working on several system and sub-system level tasks involving inter-task interfaces, interface management and vendor systems interfaces. Mr. Poupore's involvement in these tasks has included system design, development and testing. He was primarily responsible for the design and development of the Shared Memory Management system. He has also been involved with a real-time debugger, the sequencer software and a real-time, no-wait I/O system. Mr. Poupore holds a B. S. degree in computer engineering from Rochester Institute of Technology.

INTERFACES AND THEIR MANAGEMENT IN A LARGE ADA PROJECT

Walter E. Zink, Senior Systems Engineer
and
Richard E. Poupore, Systems Engineer
CAE-Link Corporation, Binghamton, NY

INTRODUCTION

This paper provides an overview of the management of *interfaces among software elements* and its *significance* in the overall life cycle process. To do this, the paper begins with requirements imposed on the B2-Aircrew Training Device (ATD) and a view of the simulation computer environment. This is followed by the issues associated with interfaces and the project's requirements. An interface management system is discussed. This system includes a tool to manage interfaces and generate interface data movement software. The system also includes the design of the generated software. The paper concludes with a discussion of the benefits derived from this interface management system.

REQUIREMENTS

The B2-ATD is a high fidelity training device. The training device is required to provide not only a realistic copy of the cockpit but also a *realistic training* environment. To provide this, a substantial amount of Ada code is needed. The B2-ATD currently has almost two million lines of code.

The B2-ATD was one of the early Air Force Ada projects. The *Air Force felt* that *traditional simulation* architecture would not be sufficient. They required that a structural model, as defined by Carnegie Mellon University's Software Engineering Institute (SEI)[1], be used for the software architecture. Part of the required structural model is a software data bus. As interfaces move across the data bus they must remain uniform with respect to time and each other. A Computer Software Component (CSC) must be able to retrieve data from another CSC in a consistent manor. A CSC is defined as a simulated system such as engines.

As with most military projects, the Air Force placed the requirement that all CSCs be independently testable. An engineer should be able to write tests in a development environment and use those tests on the simulator. The engineer should also be able to insert the CSC into the simulation load and be able to test it when not all of the CSCs on which it depends are in the load.

The Air Force also required the ability to keep the software current with the B2 air vehicle. If a system is added or modified in the air vehicle, a corresponding change must be made to the simulator. These changes need to be made in a fast and accurate way. To support this, interfaces must be easy to update.

The B2-ATD is also required by DOD-STD 2167A[2] to provide documentation of interfaces. An Interface Design Document (IDD) that *reflects* interfaces of the *simulator* must be provided.

ENVIRONMENT

Hardware

The main computational engine consists of five systems, each with four processors. Each system has a base rate. The processors in that system run at the base rate or some even submultiple of that rate. The systems run at different base rates depending on the simulated systems running on them or the peripheral devices hanging from them. The computational engine is where most of the simulation software executes. It also contains most of the over thirteen thousand interfaces. A pair of workstations power the Instructor Operator System (IOS), the man-machine interface for the instructor. Several peripheral devices are connected to the computational engine to perform a variety of functions. These include an Image Generator (IG), a Digital Radar Land Mass System (DRLMS), interfaces to B2 On-Board Computers (OBCs) and a VME-based hardware *interface system*. The *simulation computer* complex is shown in Figure 1.

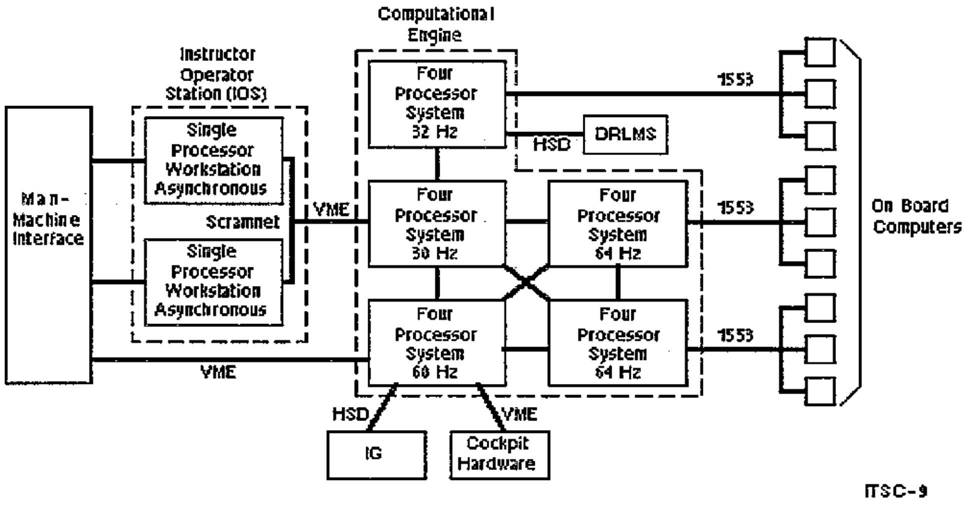
Computational Engine
Instructor Operator Station (IOS)
Man-Machine Interface
Single Processor Workstation Asynchronous
Single Processor Workstation Asynchronous
Scramnet
VME
Four Processor System 32 Hz
Four Processor System 30 Hz
Four Processor System 60 Hz
Four Processor System 64 Hz
Four Processor System 64 Hz
HSD
DRLMS
1553
1553
1553
On Board Computers
VME
HSD
VME
IG
Cockpit Hardware
ITSC-9

Figure 1 The Simulation Computer Complex

Software

The Ada architecture provides a unified and consistent basis for developing software which operates in a complex hardware environment. It is consistent with SEI's Air Vehicle Architecture[3]. It is comprised of several classes of "elements", each element performing a specified function in the overall makeup of the system and subsystems. It provides stringent data and control flow as well as the ability to test an immature or partially defined system.

The architecture is based around the autonomous CSC, its immediate environment and its relationship to other CSCs. Each CSC is comprised of object definition package(s), object declaration package(s) and a CSC control manager. Figure 2 depicts a typical CSC, its process and data flow. The figure uses a graphical notation described in "A Graphical Notation For Software"[4].

The definition package(s) provides the abstract for the CSC's objects and their control. It provides types that define the object's internal data structures. The definition package(s) also contains functions and procedures that provide the basic functionality for manipulation of the objects. The declaration package(s) provides instances of the CSC's objects. The control manager is responsible for determining how the CSC will respond to any given simulation state and the update of the CSC's objects. It invokes the operations in the definition package(s) and thereby controls the execution of the CSC.

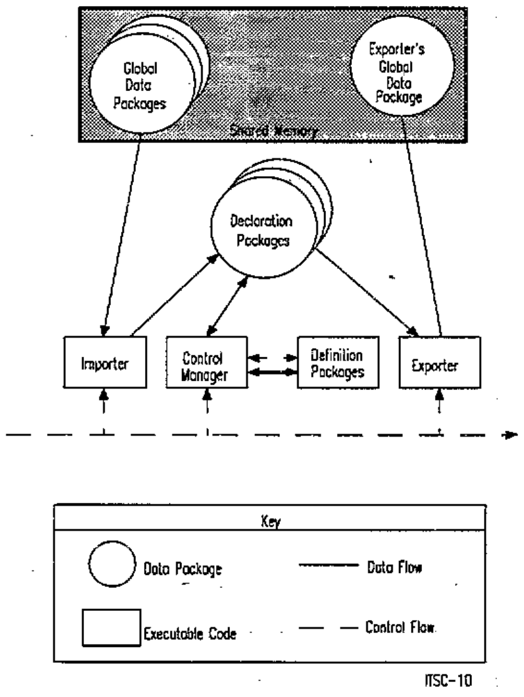

Figure 2 A Typical CSC

Wrapped around the CSC are the structural model's interface elements. These interface elements are not part of the CSC, they are the software data bus. Prior to execution of a typical CSC the importer element is called. The importer consumes data from other CSCs by transferring that data from a shared memory region to the CSC's local objects. After the CSC control manager runs, the software sequencer invokes the exporter element. The exporter transfers data produced by the CSC from the local objects to the shared memory region for other CSCs to import.

ISSUES

The first issue addressed is the software data bus. It must support all of the requirements imposed on the B2-ATD. The data bus must not interfere with the CSC software. It must copy data from one CSC to another and allow those CSCs to be independent of each other.

Since the simulator has many processors, the data bus requires intermediate data storage in a shared memory region. The structure of this data storage area is very important. If the data is placed in one data package for the entire simulator, Ada requires that the entire data bus be recompiled each time an interface is changed. On the other hand, interface data could be packaged one object per Ada package. This minimizes recompilation but imposes a pair of problems. The first problem is one of providing Configuration Management (CM) for all of these Ada units. The second problem is that many Ada compilers have limitations as to the amount of withing that an Ada unit can have. One of the ATD's export elements has over one thousand interfaces associated with it. At one object per data package, the export element would require over one thousand with statements which would break most compilers.

The software data structure should also allow the interface elements to move data at maximum speed. It must also provide maximum utilization of memory when storing data in shared memory.

In a single thread process the execution of any given software component can be precisely predicted with respect to the execution of all other software components. Therefore, the availability and integrity of interface data can be guaranteed. The issues of data consistency and integrity in such an environment is not a problem. Since the B2-ATD is a multi-thread, multi-rate environment, these issues are a major concern. An interface object can be updated by the producer at the same time a consumer is trying to read that data, resulting in the loss of data consistency and integrity. The software data bus must be built so that this does not happen.

Decoupling is a major issue stemming from the requirements of CSC autonomy and independent testability. An engineer needs to be able to develop software with external interfaces when the external software does not yet exist. An even harder problem is how to independently test software when the external interface is being driven. Decoupling the software modules provides maximum reusability and flexibility, both desirable qualities.

The requirement to keep the ATD current with the air vehicle magnifies the issues concerning rapid interface updates. The need to provide a means to verify interfaces and provide concordance is in the forefront. How is that information provided and how is it maintained?

INTERFACE MANAGEMENT APPROACH

Managing interfaces is not just providing a solid software architecture. An entire system must be developed to support the interfaces. The Shared Memory Management (SMM) system provides that support on the B2-ATD. The SMM system consists of two parts, an Interface Management Tool (IMT) and Software Interfaces (SI). The Interface Management Tool manages a data base of interface information and the generation of the connection manager software. It supports the update of interfaces and provides reports and documentation of interfaces. The Software Interfaces subsection of Shared Memory Management is the connection manager software generated by the Interface Management Tool. It runs on the simulator and is the software data bus.

Interface Management Tool

The IMT is actually a tool set. The heart of the IMT is the Interface Update Processor (IUP). All other tools, with the exception of the Interface Design Document (IDD)/interface report generator, support the IUP. The IMT is layered over a relational data base. The relational data base is used by the IMT to store interface information for rapid retrieval.

Interface Update Processor

The Interface Update Processor's function is to determine interface updates, update the data base and generate new connection manager software. This process is shown in Figure 3. To accomplish this task, the declaration packages with interface updates are parsed and the interface information is extracted. The information is parsed from a combination of Ada code and comments. The types of information extracted are:

- Interface label
- Source package and object, for exports
- Destination package and object, for imports
- Object type (package and type mark)
- Initial values, for exports
- A brief description, for exports

This information allows the interface software to know where the data is exported from and imported to. It is also used in creating intermediate storage and initializing that intermediate object. The intermediate storage is discussed in the Software Interfaces section below. The interface description provides information used to document the interface. The interface label is a name given to a specific interface. It is used to make the connection between the export of the data and all imports.

The interface information need not be placed in an object declaration package. The IUP allows the information to be placed in a text file. This text file can contain the interface information for one or many declaration packages. This option is available on the B2-ATD, but is not currently being used.

The interface information found in Figure 4 is an example of information found in a declarations package. Figure 5 shows the same information as it looks in a text file. In both figures the upper case text indicates key words used to parse the files. The lower case text indicates information being extracted from the file.

In Figure 5 the initialization information for the time of day object is too long to fit on one line. The IUP allows the initialization data to be on more than one line. The ability to provide more than one line of input allows initialization of large composite types such as records of arrays of records.

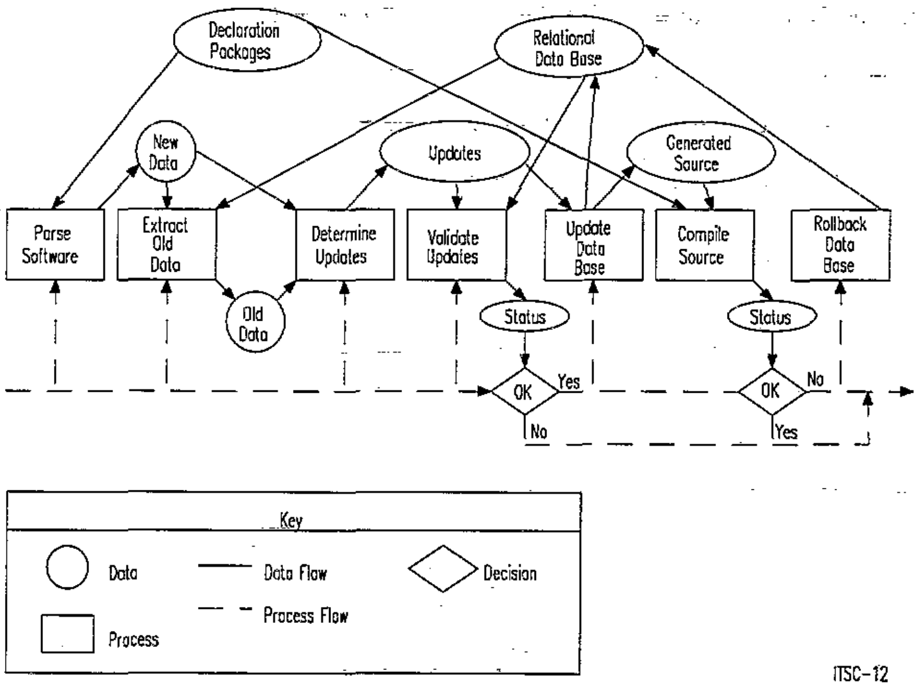

Figure 3 The Interface Update Processor

```
-->EXPORTS
-->LABEL time_of_day
   current_time : time_types.times
      := (hour=>12,minute=>0,second=>0);
-->DESCRIPTION current time of day
-->LABEL day_of_year
   current_day : time_types.days := 1;
-->DESCRIPTION current day of year
-->END
```

Figure 4 Interface Information In Source Code

```
-->PACKAGE some_data_package
-->EXPORTS
-->LABEL time_of_day
-->OBJECT current_time
-->TYPE time_types.times
-->INITIALIZATION (hour=>12,
-->INITIALIZATION minute=>0,
-->INITIALIZATION second=>0)
-->DESCRIPTION current time of day
-->LABEL day_of_year
-->OBJECT current_day
-->TYPE time_types.days
-->INITIALIZATION 1
-->DESCRIPTION current day of year
-->END
```

Figure 5 Interface Information In Text File

Next the IUP extracts interface information from the relational data base for the updated packages. With the two sets of data, old and new, the IUP determines what interface updates have been made. The updates are validated to insure that this change does not contain any detectable errors. These validation checks include the comparison of the types on both sides of the interface, insuring that unique fields of the data base are not violated and all required information is provided. It is at this time that project related rules are enforced. An example is that no export object can be deleted unless there are no imports of that data. The intent is to prevent the removal of an interface that is still required by another CSC.

Once the interface updates are validated, the relational data base is updated. With these updates in the data base, the IUP generates all import and export elements affected by this update. The newly generated data bus software and the updated declaration packages, along with any other units required for the update, are compiled against the project libraries to insure Ada compilation correctness.

In the event that the data base update has problems or the compilation check fails, the interface updates are rolled back out of the relational data base. By doing this, the IMT guarantees that the interface updates will work with the rest of the project software, at least to the point where a new executable can be linked and tested. It also insures that the relational data base is current with the project libraries.

The IUP has a software switch that can be used to generate data bus software. When this switch is used the IUP alters its process flow. The resulting process flow skips the "Compile Source" block show in Figure 3 and always passes through the "Rollback Data Base" block. This option gives application engineers the ability to do host environment testing without updating the relational data base or project libraries.

Support Tools

As stated previously, the IUP has a set of tools supporting it. The first of these tools is an interface information syntax checker. Its purpose is to insure that the interface data can be parsed from the updated units prior to submitting them to the IUP. Once an interface update is made, the software is passed through the syntax checker. If a syntax error is found in the interface information, an error message is displayed on the screen along with the line number of the error.

The second support tool is an interactive relational data base query tool. It gives application engineers access to current interface definitions. It not only allows an engineer to query the data base but it also allows the generation of custom interface reports. These reports are generated based on any or all of the data base fields. This information can be used to modifying interfaces and related software.

The information in the declaration packages is the real interface information data base. The relational data base is only used for rapid retrieval during the update, verification, software generation and report generation. In this way, standard CM tools can be used to CM the software and therefore the interfaces. With that in mind, a mass insert tool allows quick population of the relational data base. In the event a relational data base becomes corrupted, the mass insert tool can repopulate it from the CM'ed source.

The final tool is an interface definition report generator. The output of this tool is used to build a project IDD. The IDD is used for compliance with DOD-STD 2167A. It is also used by application and project engineers when designing major and minor software upgrades.

Software Interfaces

In developing the Software Interfaces, it is necessary to meet the requirement for data consistency and integrity while supporting flexibility. These constraints led to the design of the exporter and importer elements.

The exporter element is comprised of two subsections, a global memory package and an exporter procedure. The global memory package contains all of the data to be placed into shared memory, available to all CSCs. This package also contains control data which defines the state of the exporter. Figure 6 shows the structure of an exporter's global data package.

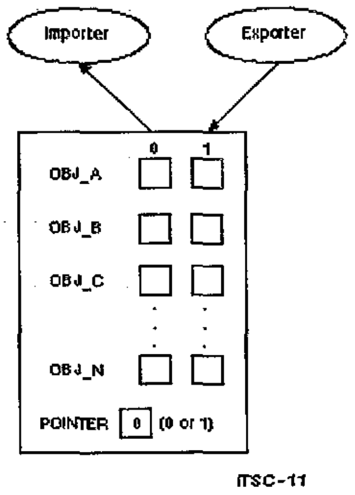

Figure 6 Data Flow In and Out of a Global Data Package

To provide data consistency, every global object in the global memory package is double buffered. One buffer is the current import object, the second buffer is the next export object. By maintaining two buffers the exporter can be writing to one buffer while an importer is reading from the other. Access to these buffers is managed through a pointer. This pointer is not an Ada access type, but an index into the buffer sets. For speed, a single pointer is used for all buffer sets.

The export procedure first determines the location of the next buffer set. It then transfers the latest data produced by the CSC into the next buffer. When completed, the export procedure updates the data pointer to point to the latest data.

The importer element contains an importer procedure. The import procedure copies the data pointers, of all exporters being imported from, to local copies. These local copies are used to retrieve consistent sets of import data. The consistency is maintained even if an exporter updates its data or pointer during the importer's processing.

A feature designed into the importer procedure is a bypass condition. The use of a bypass condition allows a control manager to turn the movement of data on or off in the importer. A bypass condition can be attached to one or more interface objects. For testing purposes, the designer may group all imports coming from a specific exporter and attach one bypass condition to this group. If the exporting CSC is not currently in the system, the importer can bypass all imports from that CSC. The interface objects can then be set to suitable default values. In addition, an emulator or an off-line development tool can be used to produce dynamic input values for the CSC. This can be accomplished without modifying a single line of code in the CSC.

The importer and exporter are structured to optimize their run-time characteristics. The emphasis is placed on speed with space as the secondary concern and lines of code the lowest priority. Development environment constraints of recompilation and relinking are also taken into account.

The exporter procedure sequencing is straight line. As stated before, the exporter first determines which buffer to store the new data in. Each export item is then move from local memory to the global memory package. Finally, the data pointer is updated. This code contains no branching and is already optimized for speed.

The importer procedure is more complex than the exporter procedure and should be optimal. The importer procedure is segmented based on bypass conditions to limit the number of "if" statements in the code. All objects without an associated bypass condition are likewise grouped together. With cache in mind, imports inside the "if" statements are grouped on an exporter boundary.

BENEFITS

In the area of testability, the architecture allows total CSC autonomy. With a local copy of the interface data, engineers can independently test their software. In the early phase of development a test driver can be developed to drive the inputs. As the overall system matures and other CSCs are added, the import bypasses can be used to turn off inputs from systems that are not yet available. Default data or a scaled back version of the test driver can be used to provide input for these interfaces.

These features also lead to simplifying the job of keeping the simulator current with the air vehicle. The local copy of interface data and the import bypasses allow a CSC to be modified to accept inputs or provide outputs to a new CSC. This new CSC may not be in the simulation software set. The existing CSC can be updated and the test driver used for regression testing. When the new CSC is added, the bypasses can be turned off allowing the data to be imported.

The interface data query tool provides valuable information in making both small and large updates to the simulation software. By using the relational data base, detailed reports can be generated to show the interaction of CSCs. This information can be used for load balancing, CSC upgrades and other similar enhancements. These reports also fill the requirement of providing a DOD-STD 2167A IDD.

The IUP provides many benefits. It allows an application engineer to update an interface by modifying a CSC's declaration package. The engineer then runs the IUP which validates the updates and generates the changes to the connection managers. This process allows easy, rapid updates to interfaces. The IUP takes a process that once took two or three days and reduces it to minutes.

By using the IUP a uniform method of moving data between CSCs has been obtained. The method of moving data can be easily changed across the simulator. A compiler upgrade may cause a rethinking of the interface software structure. A change of global data structure or the grouping of data by type may lead to execution optimization. The B2-ATD currently has over two hundred connection managers. To make a simple change to the design of the data bus software by hand would require a huge amount of time. Changing the data bus software generator takes man-days, not man-weeks.

The primary concern is data consistency. The use of double buffering and a single data pointer for an exporter has achieved that. This method allows the exporter to update the data at the same time as an importer is importing. Since the importer copies the pointer to a local storage location, the exporter can update the pointer during the importers execution.

CONCLUSIONS

As projects grow in size and complexity, the need to start managing interfaces at the earliest possible time is paramount. Interface management can no longer be viewed as a Hardware Software Integration (HSI) activity.

A software tool to automate interface management is a necessity. This tool must be able to do type checking, interface verification, interface concordance and provide a myriad of reports. In order to insure the integrity of the interfaces, this tool must have the ability to generate the interface connection software. It must also be able to generate a project level IDD. The tools for managing the interfaces must be easy to use and be reliable.

The interaction of the software architecture and the interface methodology must be orchestrated. The two need to play together to provide a software environment that is conducive to software development. An engineer must be able to rapidly affect an update to the software. That update must be of the highest quality and reliability. Only then can the claim be made that the software is truly maintainable.

While rapid updates and maintainability are required conditions, the interface software must accomplish its job. It must maintain data consistency and must do so in an optimal manner.

The SMM system allows an application engineer to alter interfaces by changing information in their software. An engineer simply tags an object with an interface label and thereby makes a connection to the producer or consumer of that data. This is done without knowing where the data comes from or where it is going to.

REFERENCES

1 "Structural Modeling White Paper" Software Engineering Institute Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, March 1992 draft

2 "Military Standard Defense System Software Development", DOD-STD-2167A 29FEB88

3 "Structural Modeling Guidebook" Software Engineering Institute Carnegie Mellon University, Pittsburgh, Pennsylvania 15213 January 1993 draft

4 "A Graphical Notation For Software", William S. *Bennett, 1991, Marcel Dekker, Inc., New York, Basel,* Hong Kong