Design and Implementation of the Tropospheric Emission Spectrometer Science Data Processing System Framework

Stephen H. Watson
Akbar A. Thobhani[1]
Jet Propulsion Laboratory
4800 Oak Grove Drive, M/S 169-315
Pasadena, CA 91109
818-354-6675
stephen.h.watson@jpl.nasa.gov
akbar@itginc.com

Benny B. Chan
Raytheon ITSS
299 N. Euclid Avenue
Pasadena, CA 91101
benny.b.chan@jpl.nasa.gov

Abstract— The Tropospheric Emission Spectrometer (TES) Science Data Processing System (SDPS) is being designed and implemented using a Framework to encapsulate common components, provide abstract interfaces to hardware-dependent features, increase maintainability and greatly reduce the burden on science algorithm development teams. The requirements and architectural design of the TES SDPS Framework were initially presented at the IEEE Aerospace 2000 conference in *A Framework-Based Approach to Science Software Development* (Larson, *et al.*). This paper reviews the Framework, outlines the detailed designs derived from the architectural design, and discusses the major components which have been implemented to date. We also discuss various tools and techniques used during implementation, including our extensive use of the Unified Modeling Language (UML), round-trip engineering, and the inclusion of 3rd-party software libraries. Finally, we present our observations on the processes and results to date, and our near- and long-term future plans and goals for the TES SDPS Framework.

TABLE OF CONTENTS

[1] Now at Investment Technology Group, Inc.

1. INTRODUCTION

The Tropospheric Emission Spectrometer (TES) is a Fourier Transform Spectrometer scheduled to fly on the Earth Observing System (EOS) Aura spacecraft in June 2003. The project is managed by NASA's Jet Propulsion Laboratory (JPL). With a nominal six-year mission, the TES instrument will provide the world's first three-dimensional global data set of tropospheric ozone and its precursors. Data processing activities are planned to continue through at least June 2012, and will likely extend considerably beyond that time.

TES is a state-of-the-art instrument in terms of its combined performance, resolution and operational capabilities. The instrument will produce over 8TB of raw data, and an additional 25 TB of processed data each year. Evolution of the algorithms, and the production software that implements them, is expected to continue throughout the mission, and accelerate in the first months after launch. Data processing will be performed by a production facility located in Pasadena, CA. The TES Science Investigator-Led Processing System (SIPS) will be designed, built and operated by Raytheon Information and Science Systems under contract to JPL.

The TES project team is responsible for developing and delivering processing software to the production facility, and for providing operational support for data quality monitoring and anomaly investigation. The data products produced in the SIPS will be delivered electronically to the NASA Langley Research Center (LaRC) Distributed Active Archive Center (DAAC), an element of the EOS Data and Information System (EOSDIS).

The production environment is a UNIX-based planning and scheduling system. The system places few code-level restrictions on delivered science software. However, the plan to operate the facility round-the-clock utilizing only prime shift staffing places significant requirements for automation and reliability on the science software.

Scientific research, software development, and operational support activities will be based in the Science Computing Facility (SCF) located at JPL. Past experience with remote sensing project suggests that unforeseen events in instrument calibration and on-orbit algorithm performance will necessitate the rapid development of many new applications. The ability to respond to these events in a timely manner and within budget is a major goal of the system design approach.

As part of the EOS program, TES is required to deliver all data products to the EOSDIS, this places additional requirements on the system in terms of data formats and metadata production. In order to reduce cost, TES is considering the use of a number of third party software packages for inclusion on the delivered system. These include NASA-supplied toolkit routines for data format, metadata, ephemeris, geolocation and mapping functions, freely available libraries for numerical processing, linear algebra support, and command line processing, and several commercial, off-the-shelf software packages, including a commercial database.

The extreme reliability, maintainability and reusability constraints placed on the TES software development effort, coupled with the realities of NASA science budgets both now and in the future, the project explored ways to reduce development costs while ensuring high compliance with those constraints. Further, it became clear that considerable new functionality would need to be added throughout the project lifecycle in response to science team discoveries. One key component of meeting these goals was the adoption of a framework for software development and science algorithm implementation.

The decision to adopt a framework-based approach was a strategic one, intended to fulfill the need for reduced cost, and a more robust, maintainable system. Along with the decision to develop a framework, the project adopted an object-oriented (OO) design approach and selected the C++ language. The framework decision was thus part of an overall strategy to leverage OO technology and modern approaches to reuse and development.

At the 2000 IEEE Aerospace Conference, Larson, Watson and Thobhani [1] presented the architectural design and implementation plan for the framework subsystem. This paper will discuss the current status of the TES SDPS Framework implementation, the architectural and detailed design issues which arose during the last year's development, and near- and long-term plans for the completion and delivery of the Framework. We will also discuss our experiences with object-oriented design techniques, the Unified Modeling Language (UML), and various tools and 3rd-party packages.

2. OVERVIEW

The current design concept for the production software system is shown in Figure 1. The boxes labeled "PGE's" represent units of execution that include staging, processing and destaging steps within the production environment. The actual data flow is shown in Figure 2. The processing flow in the SCF (not shown) is more complex, but less formalized. It is expected to comprise a few dozen applications in the year 2002 time frame. As discussed above, the need to develop SCF applications rapidly is a major driver in our decision to develop a framework. However, we will focus much of our discussion on the production system, as it is more well-defined, and highlights the major features of the framework satisfactorily.

The total system size is expected to be roughly 500,000 lines of code, including comments. We include comments in system sizing, as we have found that they represent a significant portion of the development work, as well as comprising one of the most important system documentation records.

The production system design is based on an EOS-mandated demarcation of processing into levels. The basic unit of processing is a 1-day data set (also known as a global survey cycle (GSC)), which is received every two days (the remaining day is reserved for special science observations, which will not be discussed here). For details of the specific level processing requirements, the interested reader is referred to another paper in this session, *Overview of the Tropospheric Emission Spectrometer Ground System Development*, by S. Larson and R. Beer.

The framework is being developed to support the system-wide functionality summarized in Table 1. As of this writing, the core elements necessary to support basic end-to-end instantiation of *Product Generation Executables* (PGEs) and data flow simulation have been designed, implemented and integrated. This includes the PGE infrastructure, parameter handling, exception handling and File I/O class hierarchies. We are currently in the process of designing the metadata and logical data object hierarchies. Design and implementation of process control class structures is awaiting a decision as to the ultimate SIPS architecture. Finally, we have successfully reverse engineered (via Rational Rose's toolset) several 3rd-party packages, such as RogueWave's tools.h++, math.h++ and lapack.h++ packages.

The complete set of packages being provided by the TES SDPS Framework is shown in Figure 3.

PGE Infrastructure	An application skeleton for a PGE executable.
Parameter Handling	Support for parameters from files, environment, command lines.
Exception Handling	Classes and conventions for handling exceptions.
File I/O	Support for I/O to/from all files in the system. Format support for HDF-EOS, HDF5, ASCII and native binary file types.
Data Types	I/O support in terms of high-level data types. Support for specific data product file organizations.
Metadata	Support for EOS standard metadata output.
Database Support	C++ API support for commercial database application.
Multiprocessing/Multithreading	Threaded application interfaces.
COTS	Support for linear algebra, mathematical functions, specialized optimization and Fourier transforms, other functions. Additionally, other 3rd-party packages may be included.

Table 1. Summary of Framework Requirements

The following section details the design and implementation of the core framework components which have been implemented to date, as well as those which are currently being developed in anticipation of our first internal end-to-end system test (currently scheduled for early in 2001).

3. Framework Components

PGE Infrastructure: Algorithm/Environment Interface

The primary interface between the science algorithm and the SDPS environment is through a set of objects called the Environment, the TES *Algorithm Interface (TAI)* and the *TES Algorithm.* The environment class provides a means to isolate environment-specific elements, including interfaces directly to the operating system, and remove the possibility of the science algorithm development teams creating system-dependent code. The purpose of the TAI is to encapsulate and abstract all information from the external environment, and make it available to an algorithm in a clean, transportable manner. Ideally, this architecture should completely isolate the algorithm itself, and consequently the algorithm developers, from any system-specific details. Additionally, it provides a clean, portable and extensible interface between the algorithm and those items dependent upon the operating environment, such as files, logs, process control, etc. One view of the Environment class hierarchy is shown in Figure 4.

It should be noted that in many cases, we have multiple diagrams for the design and implementation of our class hierarchies. Indeed, this is a powerful feature of our UML modeling tool, and we use it extensively to evaluate designs in different contexts. In this paper, we will usually present only a single view of a component or components.

The Environment class hierarchy is a singleton object (cf. Gamma, *et al.,* [2]) whose primary tasks are to pass command line information regarding parameters to the parameter handling object, and to execute the TAI. Additionally, the Environment class provides a means for the algorithm to output messages to a console or operator.

The TAI executes two major tasks: instructing the parameter handling mechanism to begin parsing (starting from the command line); and it then creates an instance of a specific science data processing algorithm and executes it, providing all necessary information about the system and the run-time parameters needed by the algorithm.

Instantiated within the TAI is a derived class of type *FW_CTES_Algorithm.* This derived class is highly specific and usually extremely computational. The Algorithm component is based on the Bridge design pattern. Each level of processing will derive its own algorithm object, and use this derived class as a template or wrapper for including or developing the scientific data processing code. The base class itself provides access to those common properties and methods that are deemed essential to ensure consistent operation of algorithm objects throughout the data processing sequence. This base class supports log and error file handling, exception handling, process control, etc., via the aggregation of objects of types specific to those items. Access to system- or process-wide parameters is done via the TAI's parameter instance (itself a singleton object).

Figure 5 shows the Unified Modeling Language representation of the Environment, Algorithm Interface and Algorithm classes.

Parameter Handling

The Framework must be able to handle a virtually unlimited number of parameters that may be specified as inputs to any given algorithm. These parameters are defined in a separate parameter definition file for each level, as well as in coded default types for some parameters that may used in more than one algorithm.

After parsing the definitions of all the specified parameters, the actual values may be declared in any number of places. The framework provides for the instantiation of a singleton object which contains a parameter block capable of parsing these values from the variety of locales in which they may be set. These parameter values can be in a parameter file, as part of the operating system environment, specified on the command line or set as default values by the parameter definition itself. The parameters may be of virtually any type, including compound types.

We currently provide support for the complete range of numeric and alphanumeric types, as well as Boolean, strings, and several types we term "checkbox" and "radiobutton" parameters (due to their similarity to GUI-based radiobuttons and checkboxes, which allow one of many and several of many Boolean values to be turned on, respectively). Additionally, the parameter handling classes can support nested parameter blocks to a predefined depth of four nested levels. This was implemented to support separation of parameters by level, and to prevent name collisions between disparate teams working simultaneously.

As reported in out previous publication [1], we are heavily employing the use of templates to provide consistent interfaces to all parameter types. Additionally, templatized functions provide a simplified interface with a guaranteed correct return type as determined by the algorithm at run-time. Figure 6 shows the current implementation of the parameter-handling hierarchy.

Exception Handling

Although fairly straightforward, a complete, logical exception-handling strategy is ultimately of critical importance in the TES SDPS, due to the batch-oriented, 24-hour processing scheme which will be necessary to process the massive amounts of data over the instrument lifetime. In conjunction with system engineering, the Framework team has developed a robust, tightly constrained exception-handling process.

A consistent, manageable set of error codes and algorithms for dealing with exception handling has been devised as the key component of this portion part of the Framework design. We have developed a hierarchy which parses in a standard C++ header file containing the definitions of the error codes and their symbolic constants, along with a descriptive comment about each one. By parsing these codes directly from the level-specific processing header files, absolute consistency between the exception-handling mechanism and the algorithmic implementation is maintained at all times, *i.e.*, every exception and its descriptive message must be identified during algorithm implementation and must, therefore, pass our code review and testing process.

In addition, we have defined a strict protocol for the use of exceptions throughout our system. First, unless authorized by the cognizant engineer and the element manager, every module must contain a *try-catch* block. The simple inclusion of these blocks throughout the system is designed to force designers and implementers to evaluate thoroughly the types of exception-handling required by each method. Secondly, each exception has a clearly defined severity level, ranging from OKAY (meant to imply that the exception can be dealt with immediately with no ill effects on processing) to FATAL (execution terminates immediately). All exceptions are logged immediately upon instantiation, and the may have their severity levels increased by an exception-handler, but not decreased.

Figure 7 shows the current implementation of the exception-handling hierarchy.

File Handling

One of the most important areas that the Framework must deal with is that of file handling. It has always been a primary goal of the TES SDPS Framework to provide algorithm developers with a file mechanism that is completely transparent, incorporating complete independence of algorithms on underlying OS' file handling mechanism. In fact, the requirements to support HDF-EOS via a toolkit for file access is driving a very complex file handling structure. Additionally, the nature of the data to be stored implies a large amount of additional complexity.

As reported in [1], the architectural design is based on a combination of several design patterns [2]. The logical files, which provide the interface to all subsystems, are a straightforward dependency from a base class called *Logical File*. Logical files correspond to the needs of any given level-specific algorithm, such as Interferogram data files, raw data packet files, etc. The logical files provide an interface toaccess Logical Data Objects from different Physical Files.

Physical files encapsulate environment and format-specific details of actual data storage, isolating these details from the application programmer. In some instances, such as HDF format files, data may span more than one actual disk file. It is the responsibility of the Physical File hierarchy to handle such matters, perhaps using 3rd-party toolkits in some instances. The Physical File hierarchy is based, therefore, on the Adapter pattern. The relationship between these two sub-components is maintained via a set of Bridge patterns, one bridge for each derived logical file type.

The top-level interaction between all of these file hierarchies is shown in Figure 8.

This diagram presents the basic hierarchy with an example logical file, *FW_CTES_Data_File*, which is derived from the logical file class. It aggregates a pointer to the base class for physical files, *FW_CTES_Data_Physical_File*, which is inherited from the physical file hierarchy. The actual class of the physical file, in this case either HDF or binary, is determined at runtime. However, through the judicious use of virtual functions in the base class hierarchies, we are assured that the proper methods are called during read and write operations.

Data files are connected to our internal data structures (what we term *Logical Data Objects*) using layout object which describe how internal data is structured in the physical files. As the next subsection shows, there is a relatively simple way to implement the relationship between actual data objects and the physical files into which the data is written.

To date, the file handling hierarchies have been the most problematic portions of the framework. This is due in large part to the complexities of the EOS-mandated Hierarchical Data Format for all products which are to be archived at the Langley DAAC, but it is also due in part to the requirement to isolate, to the maximum extent possible, the algorithm developers from the details of any single file format. The diagrams shown here are only a small sampling of the complete UML diagrams for the file handling package, and a complete description is beyond the scope of this paper. Nevertheless, the essential portions of these class hierarchies are in place at this time, and are operationally ready for simple file handling operations. Additionally, we are indebted to NCSA's publicly available C++ API for HDF, which helped us avoid direct calls to the HDF toolkit at the lowest levels of our implementation.

Logical Data Objects

For some types of data, the framework will provide the class hierarchies which support those data types. Concurrent with the development of file handling and I/O mechanisms, we determined the need to develop a rich hierarchy of *Logical Data Objects*. These objects will need to interface with the file structure in order to read and write themselves to the data files. Examples of high-level domain-related data types that must be supported are *scan*, *focal plane*, *interferogram*, and *spectrum*. These may be singular or combined in various ways, including multiple instances of any one type or combinations of types. It is highly likely that additional types will be required as science algorithm development and scientific data analysis proceeds during pre-launch and operational phases. Therefore, the architecture must provide an easily extensible model. Logical data objects are therefore based on the Composite design pattern.

Now, in order to provide read and write capability to the various file formats which we are supporting (nominally HDF5 or the forthcoming HDF-EOS based on HDF5), the Framework provides that every logical data object has an associated layout. This layout specifies how a data object's read/write calls are to be interpreted by the file component, thus in effect providing a map from the logical structure to physical structure. The layout for a specific data object is provided by a class factory object, which constructs the layout for a given file type and data object. As a data object is traversed hierarchically for either input or output, layout objects are retrieved from the factory and placed on a stack. The stack is successively popped as write operations (including opening and closing HDF groups) are completed. These relationships are shown in Figure 9. Additionally, it should be noted that Logical Data Objects in fact write themselves to output files (or read themselves from input files) using the Data Access Interface. In fact, our main obstacle in File Handling was to minimize data and file format coupling. We accomplish this via Data Access Interface.

At this writing, we are beginning the work of creating the complete initial set of logical data objects and their layout objects. Our initial testing of the operation of some basic data objects during read and write operations has shown that this is indeed a viable method, and that extensions to additional data types or file formats will be readily accomplished. Indeed, the need to be able to easily add file formats is paramount, as the time until first operations after launch is nearly over 3 years from this date, virtually guaranteeing file format updates and revisions. That, coupled with the long mission lifetime (in excess of 5 years) implies that we will be adding and/or changing file format support for some time to come.

Metadata Support

The Framework team has recently begun the effort to create metadata associated with standard data product files. These metadata are, in fact, required by EOS to accompany standard data products, to provide users with search and order capabilities. It is anticipated that by the time of publication of this paper, we will have complete support for all types of metadata required by the SDPS.

There are three types of metadata that must be supported by the Framework. Two of these types, inventory and archive metadata, are supported by the EOSDIS Core System (ECS) Product Generation System (PGS) Toolkit, supplied to all EOS projects by NASA. The Framework must provide a wrapper class for the Toolkit's metadata functions in order to simplify the task of application developers. Toolkit-supported metadata are documented in an interface control document between the TES project and the Langley DAAC, utilizing an EOS standard format definition language. This is necessary since the ingest functions at the DAAC must be tailored to meet the TES metadata structure. The actual metadata definition file is automatically used by the toolkit to create a metadata set associated with an HDF file.

Inventory metadata include instrument identification, date and time of data acquisition, geographical location, product type and level, and data quality metrics. Archive metadata comprise a broader set of attributes that are stored within the data product files. These data are not stored in the ECS database, and thus are not available to search tools.

The third type of metadata supported by the Framework is instrument team-defined metadata. These data are not supported by the PGS Toolkit. Attributes defined by the instrument team are considerably more complex in structure than inventory and archive metadata. It is unclear at this time whether this metadata will be created and accessed via a derived metadata class, or through the use of a commercial database (see below).

Database Support

Recently, the TES SDPS team has concluded that some form of database will be required to handle a large amount of information which is utilized to optimize the processing flow. It is likely, for example, that the level 1 processing algorithms will produce thousands, if not hundreds of thousands, of output files for a single global survey. In order to efficiently process this data, the level 2 algorithms will need to collect the appropriate files for many individual portions of this data, perhaps within a geographic region, and process the complete data as a number of small sections utilizing multiprocessing capabilities. For reasons such as this, a database will be required.

Since database support will be used by more than one level processing subsystem, by definition this is a Framework task. Fortunately, it appears that a simple solution is at hand. The Microwave Limb Sounder (MLS) instrument team has developed a C++ interface to an Oracle database server, which appears to suit our needs. The Framework team has consistently been a strong advocate of reuse of existing code or commercial packages, and this is a prime example of the efficacy of such an approach.

Multiprocessing Support

Due to the extreme cpu-boundedness of portions of the SDPS, specifically the level 2 retrieval algorithms, it will be necessary to support both multiprocessing and multithreading. Multiprocessing is expected to be the domain of the SIPS scheduling and planning software. However, it will be the responsibility of the Framework design to incorporate multithreading support. For our initial integration and testing, however, such support is only in the evaluation stage, as we intend to defer design decisions until a firm hardware architecture plan is in place. Ultimately, we anticipate a separate class hierarchy specifically designed to provide a simple interface to the underlying thread control methods. The terminal object(s) in this hierarchy may then be included in either the abstract base class *Algorithm* or a level-specific derived algorithm object thus providing thread control support to the derived algorithm objects.

Commercial Off-the-Shelf Software and Miscellaneous Components

Finally, we have incorporated, in some cases via Rational Rose's reverse engineering capability, several COTS packages. Chief amongst these are RogueWave's Standard C++ Library, tools.h++, math.h++ and lapack.h++. This has not been without its pitfalls, as we have continuously encountered issues related to different compiler versions and software releases. This has been primarily due to varying implementations of the C++ standard by different compiler vendors, including such issues as templates, namespaces and so forth. This appears to be an on-going issue, and the eventual choice of SIPS hardware and operating system may have an even greater impact on these packages. This highlights why the Framework has a primary goal of isolating science algorithm developers from OS, hardware and support package changes. Modifying only the Framework in the event of a change to any one of these will significantly reduce the overall TES SDPS cost and risk.

4. ISSUES

Design and development tools

The Framework has been at the forefront of many of the tools used by the SDPS development team as a whole. Since the adoption of the Unified Modeling Language (UML) for all portions of the system design and implementation, Framework has been, in general, the first major subsystem to encounter new features and/or problems. We are using Rational's Rose CASE tool to develop and document the design, as well as to generate code and reverse engineer external components. In nearly all cases, the architectural and detailed designs have gone extremely smoothly. However, some limitations in Rose's code generation capabilities have caused some consternation. For example, as we stated above, we would like to enforce the use of try-catch blocks in all methods. However, it is not possible to force Rose to generate this code within what are known as "preserved regions". This would be a useful feature, as we believe that the more automated the code generation process, the less prone to failure it is. There are numerous examples similar to this which, although not outright failures, have caused some additional effort to develop work-arounds. Largely, however, the team is quite please with the tools capabilities and robustness. The single most difficult problem lies in reverse engineering or round-trip engineering in Rose, but we expect that our increased experience in the coming years will obviate some of these concerns. It is anticipated that this will be an important functionality, as we intend to keep our designs current through round-trip engineering of algorithmic changes.

Design and interface documents are currently generated automatically from the Rose model using Rational's SoDA documentation tool. This tool consists of a number of Word scripts, which can be modified through a GUI, which access the model and extract information to be placed in a document. Thus, we are not manually creating documents such as interface control documents, design documents, etc. Instead, the combination of the Rose and SoDA ensure that documentation continually remains up to date.

Requirements are tracked using Rational's RequisitePro database, which again interfaces to Word for the creation and maintenance of requirements and their trace matrices. Again, this means that the actual document which is created is merely a representation of the current requirements set, which is maintained via RequisitePro in a database.

Since Framework has been, to date, the first subsystem to go through the full cycle from architectural design to release, it is also the first one to be subjected to some newer tools for test coverage analysis and code review. We have recently acquired two of these tools, McCabe's IQ test analysis software, and Parasoft's CodeWizard coding standards analysis tool. Both of these are relatively new acquisitions. We have just begun using CodeWizard to analyze all Framework code for conformance to a set of best C++ practices and in-house coding standards. We intend to require *all* software to pass this analysis before acceptance.

We continue to evaluate and acquire appropriate 3rd-party packages for inclusion in or use by the Framework subsystem. Specifically, we are attempting to avoid the "not invented here" syndrome and to strive for lowered costs and increased robustness wherever possible.

Staffing

In our previous report, we stated:

> "Our ability to attract and retain qualified staff is the most important non-technical risk factor associated with the Framework development."

In the intervening year, the situation has not improved. In fact, what was a small pool of candidates has shrunk further due to the seemingly ever-increasing salaries offered by for-profit corporations. Indeed, one of us (Thobhani) recently left JPL for just such a position. These competitors can offer earning potential that a federally funded research and development center like JPL cannot match. Staffing will likely be a continuing area of difficulty for some time to come. Mitigation has been accomplished so far by descoping some areas, schedule extensions in others, and reprioritizing components in the remainder.

5. SUMMARY

The TES project is currently in the implementation and test phase of the development of a science software applications framework. Detailed design and implementation of the first increment were completed by the fall of 2000, and efforts are underway at integration and system test, as well as preparations for end-to-end data flow and operations testing. Additionally, detailed design on the next phase of components is beginning, with completion of those units expect by mid-2001.

Since our previous paper, several very major components have been built and testing, and are being integrated into a cohesive whole. Release to other subsystems to begin their production code development is imminent, and we are confident that the decision to implement a Framework-based system will contribute to a significantly reduced cost with an increased robustness.

Additionally, we are beginning to see signs that the Framework may be reusable by other instrument teams in the future. Such re-use would be a major factor in future instruments' cost reduction efforts.

6. ACKNOWLEDGEMENTS

The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

The authors would like to acknowledge the valuable contributions provided by Ken Scott, Steve Larson, Brian Morrison, Steve Friedman and Doug Shepard to the Framework analysis, design and implementation. Additionally, the Framework team recognizes the contribution of Mark Echeverri in the development of the C++ API to Oracle for MLS-AURA, and the work by NCSA on the C++ API to HDF 5.

6. REFERENCES

[1] Steve Larson, Stephen Watson and Kalyani Rengarajan, "A Framework-Based Approach to Science Software Development", *IEEE Aerospace 2000 Conference Proceedings*, March 18-25, 2000.

[2] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides and Grady Booch, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.

[3] Bjarne Stroustrup, *The C++ Programming Language*, 3rd Edition, Addison-Wesley, Reading, MA, 1999.

[4] HDF 5 Documentation, at http://hdf.ncsa.uiuc.edu/HDF5

[5] *HDF-EOS Library User's Guide, Volumes 1 & 2,* NASA Goddard Space Flight Center document # 170-TP-500-001, Greenbelt, MD: NASA GSFC, 1999. Also available at http://edhs1.gsfc.nasa.gov/waisdata/sdp/pdf/tp17050001.pdf and http://edhs1.gsfc.nasa.gov/waisdata/sdp/pdf/tp17050101.pdf

[6] Marshall P. Cline and Greg A. Lomow, *C++ FAQs*, Addison-Wesley, Reading, MA, 1994.

Stephen H. Watson is a Senior Engineer at NASA's Jet Propulsion Laboratory (JPL), and is currently the Cognizant Design Engineer for the Tropospheric Emission Spectrometer (TES) Science Data Processing System Framework. Prior to joining the TES team, he worked at Magellan Corporation developing Global Positioning System receiver and post-processing software, and was responsible for managing a multinational software team. He was previously employed at JPL as a software engineer specializing in scientific data visualization, supporting a variety of planetary and earth science missions. Mr. Watson received a Bachelor of Science in Mathematics and a Master of Science in Computer Science from Arizona State University.

Benny B. Chan is a Software Engineer on Tropospheric Emission Spectrometer (TES) Science Data Processing System Framework at NASA's Jet Propulsion Laboratory (JPL) / Raytheon. Mr. Chan was previously employed at Gencorp Aerojet as a software engineer developing and testing mission and support software for Ground-Based Infrared Data Processing and Engineering Support for Missile-Detecting Satellites (SBIRS Low) and Transportable Satellite Data Processing System (JTAGS). Mr. Chan received a Bachelor of Science in Computer Science from California Polytechnic of Pomona University.

Akbar A. Thobhani is a Software Engineer at Investment Technology Group (ITG) Inc. Prior to ITG, he was a Software Engineer at the NASA's Jet Propulsion Laboratory. At JPL he designed and developed TES Science Data Processing System Framework. Mr. Thobhani is also a Regional Project Manager for EduOnline.net, a non-profit initiative. He received a Bachelors degree in Computer Science and in Management Information Systems, with a minor in Computer Mathematics from California State University, Northridge.

Figures.

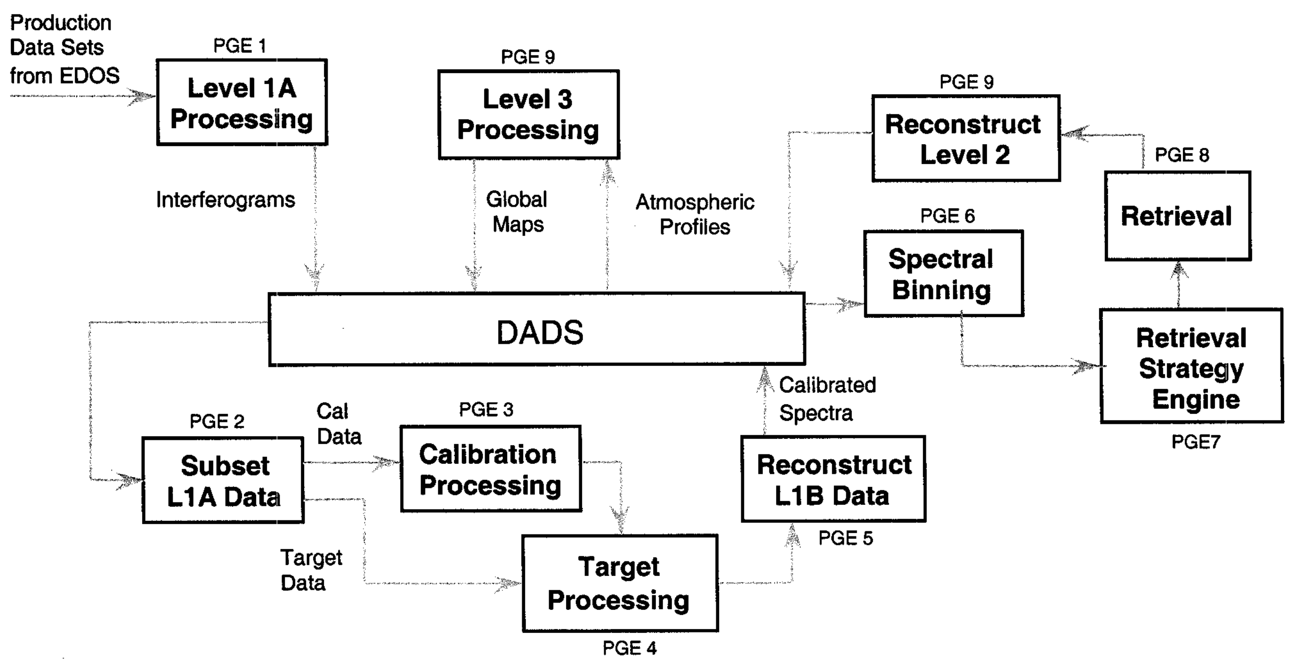
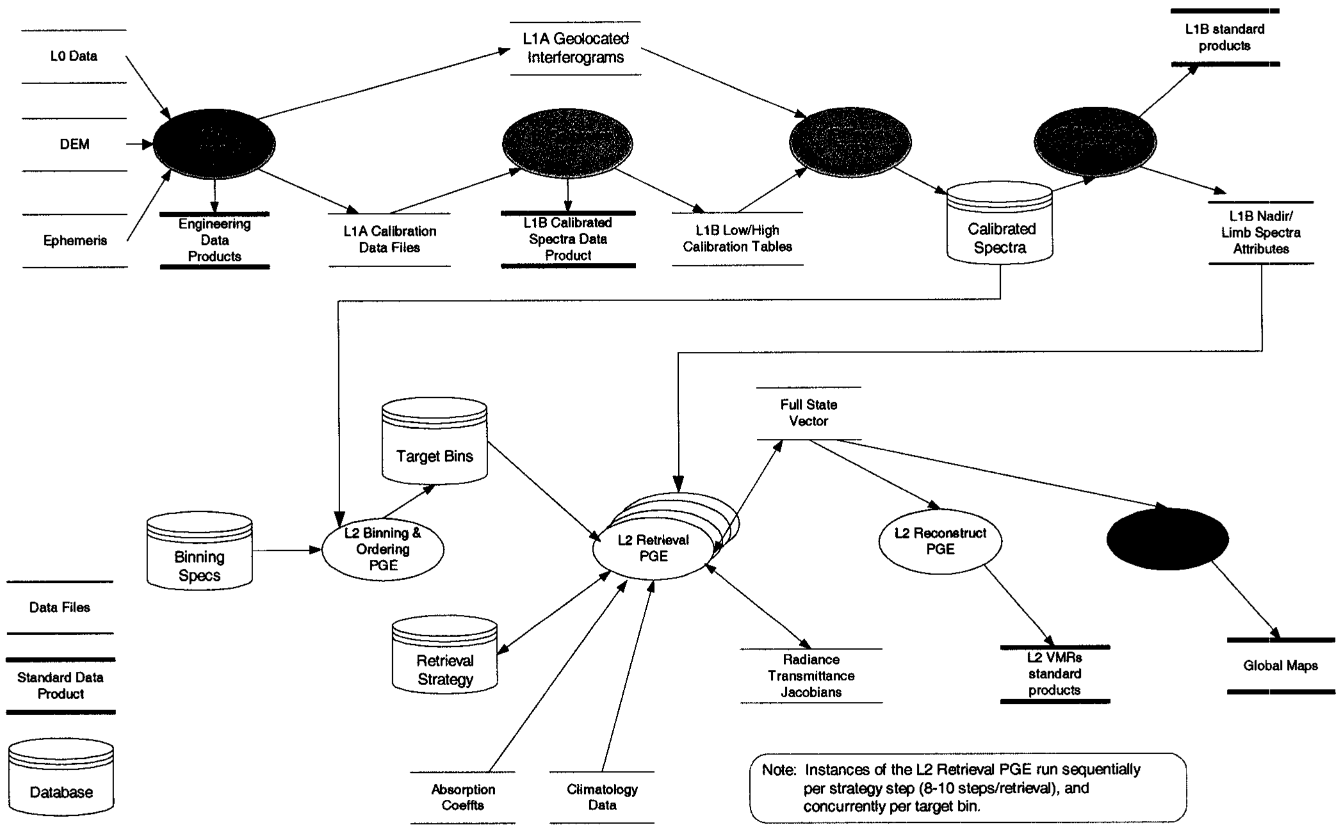


Figure 1. Data Flow Through TES Production Processing System

Data Flow Between TES SDPS PGEs



L0 Data
DEM
Ephemeris
L1A Geolocated Interferograms
L1B standard products
Engineering Data Products
L1A Calibration Data Files
L1B Calibrated Spectra Data Product
L1B Low/High Calibration Tables
Calibrated Spectra
L1B Nadir/ Limb Spectra Attributes
Full State Vector
Target Bins
Binning Specs
L2 Binning & Ordering PGE
L2 Retrieval PGE
L2 Reconstruct PGE
Data Files
Standard Data Product
Database
Retrieval Strategy
Radiance Transmittance Jacobians
L2 VMRs standard products
Global Maps
Absorption Coeffts
Climatology Data
Note: Instances of the L2 Retrieval PGE run sequentially per strategy step (8-10 steps/retrieval), and concurrently per target bin.

Figure 2. TES SDPS Data Flow

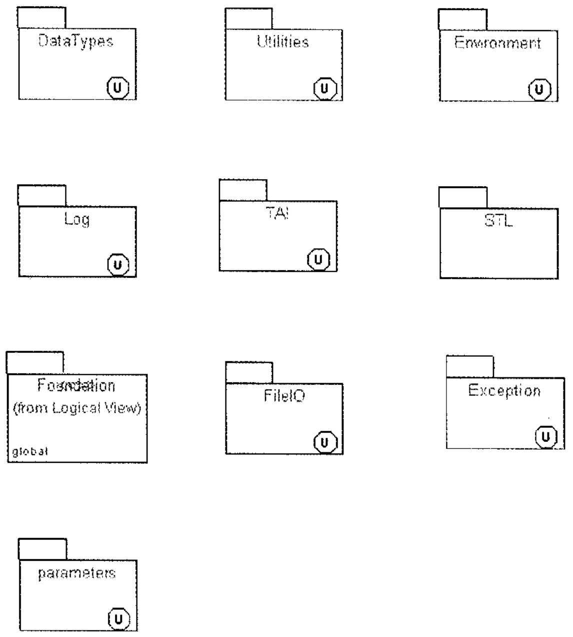


Figure 3. TES SDPS Framework Packages

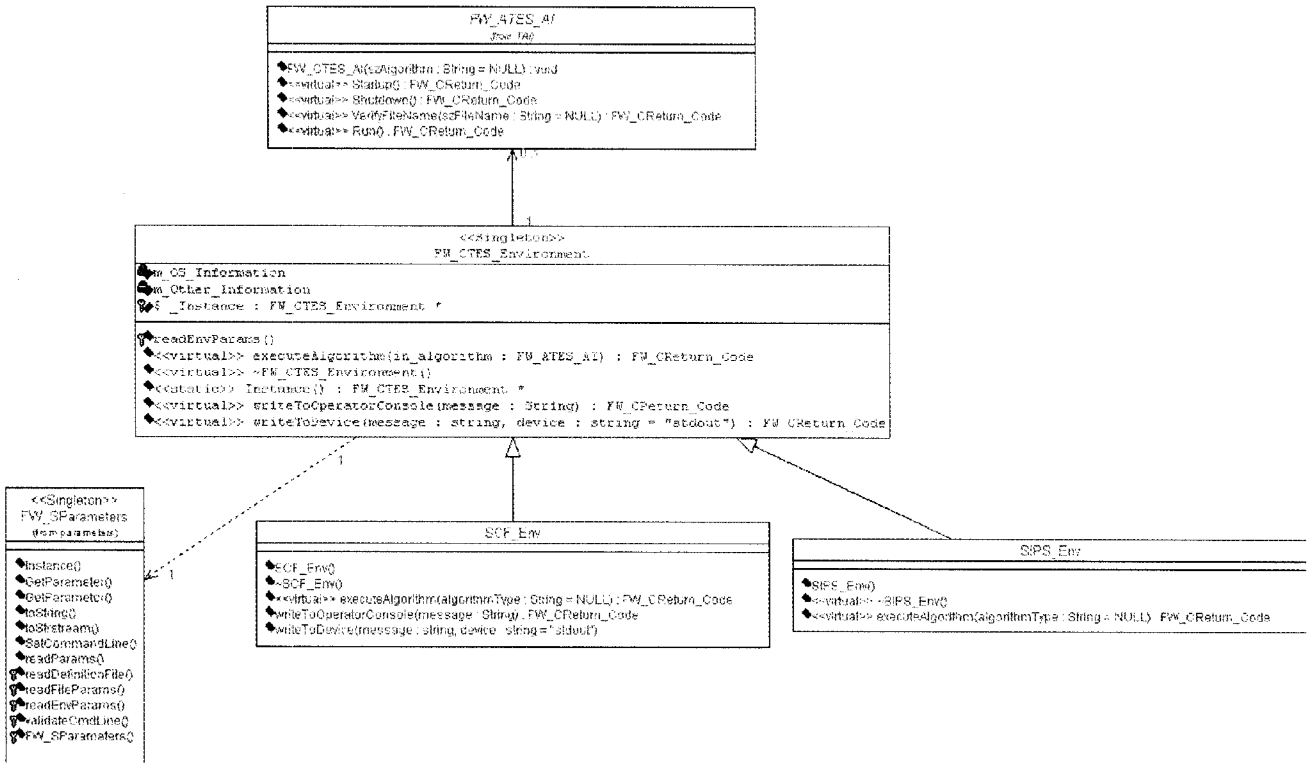

FW_ATES_AI
(from TAI)
FW_CTES_AI(szAlgorithm : String = NULL) : void
<<virtual>> Startup() : FW_CReturn_Code
<<virtual>> Shutdown() : FW_CReturn_Code
<<virtual>> VerifyFileName(szFileName : String = NULL) : FW_CReturn_Code
<<virtual>> Run() : FW_CReturn_Code
1
<<Singleton>>
FW_CTES_Environment
m_OS_Information
m_Other_Information
$ _Instance : FW_CTES_Environment *
readEnvParams()
<<virtual>> executeAlgorithm(in_algorithm : FW_ATES_AI) : FW_CReturn_Code
<<virtual>> ~FW_CTES_Environment()
<<static>> Instance() : FW_CTES_Environment *
<<virtual>> writeToOperatorConsole(message : String) : FW_CReturn_Code
<<virtual>> writeToDevice(message : string, device : string = "stdout") : FW_CReturn_Code
1
<<Singleton>>
FW_SParameters
(from parameters)
Instance()
GetParameter()
GetParameter()
toString()
toStrstream()
SetCommandLine()
readParams()
readDefinitionFile()
readFileParams()
readEnvParams()
validateCmdLine()
FW_SParameters()
1
SCF_Env
SCF_Env()
~SCF_Env()
<<virtual>> executeAlgorithm(algorithmType : String = NULL) : FW_CReturn_Code
writeToOperatorConsole(message : String) : FW_CReturn_Code
writeToDevice(message : string, device : string = "stdout")
SIPS_Env
SIPS_Env()
<<virtual>> ~SIPS_Env()
<<virtual>> executeAlgorithm(algorithmType : String = NULL) : FW_CReturn_Code

Figure 4. Environment Class Hierarchy

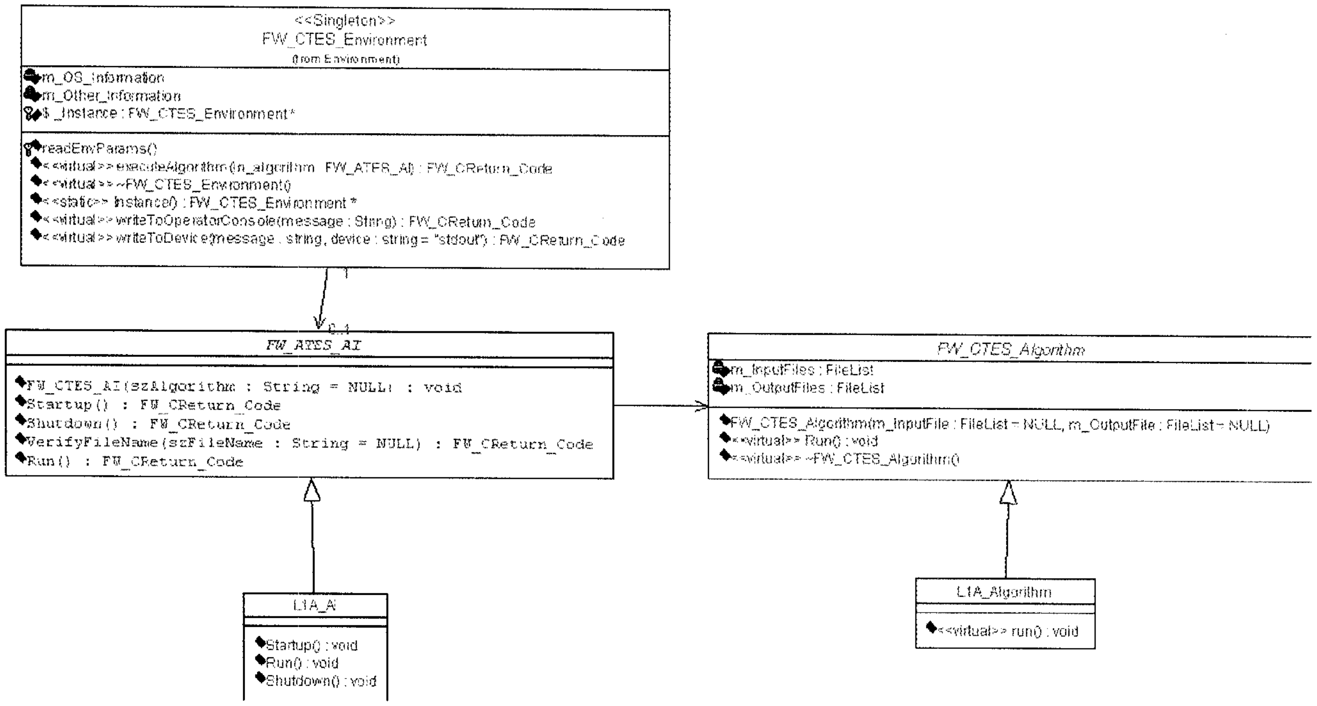

<<Singleton>>
FW_CTES_Environment
(from Environment)
m_OS_Information
m_Other_Information
$_Instance : FW_CTES_Environment*
readEnvParams()
<<virtual>> executeAlgorithm(in_algorithm : FW_ATES_AI) : FW_CReturn_Code
<<virtual>> ~FW_CTES_Environment()
<<static>> Instance() : FW_CTES_Environment*
<<virtual>> writeToOperatorConsole(message : String) : FW_CReturn_Code
<<virtual>> writeToDevice(message : string, device : string = "stdout") : FW_CReturn_Code
1
0..1
FW_ATES_AI
FW_CTES_AI(szAlgorithm : String = NULL) : void
Startup() : FW_CReturn_Code
Shutdown() : FW_CReturn_Code
VerifyFileName(szFileName : String = NULL) : FW_CReturn_Code
Run() : FW_CReturn_Code
FW_CTES_Algorithm
m_InputFiles : FileList
m_OutputFiles : FileList
FW_CTES_Algorithm(m_InputFile : FileList = NULL, m_OutputFile : FileList = NULL)
<<virtual>> Run() : void
<<virtual>> ~FW_CTES_Algorithm()
L1A_AI
Startup() : void
Run() : void
Shutdown() : void
L1A_Algorithm
<<virtual>> run() : void

Figure 5. Environment-Algorithm Interface

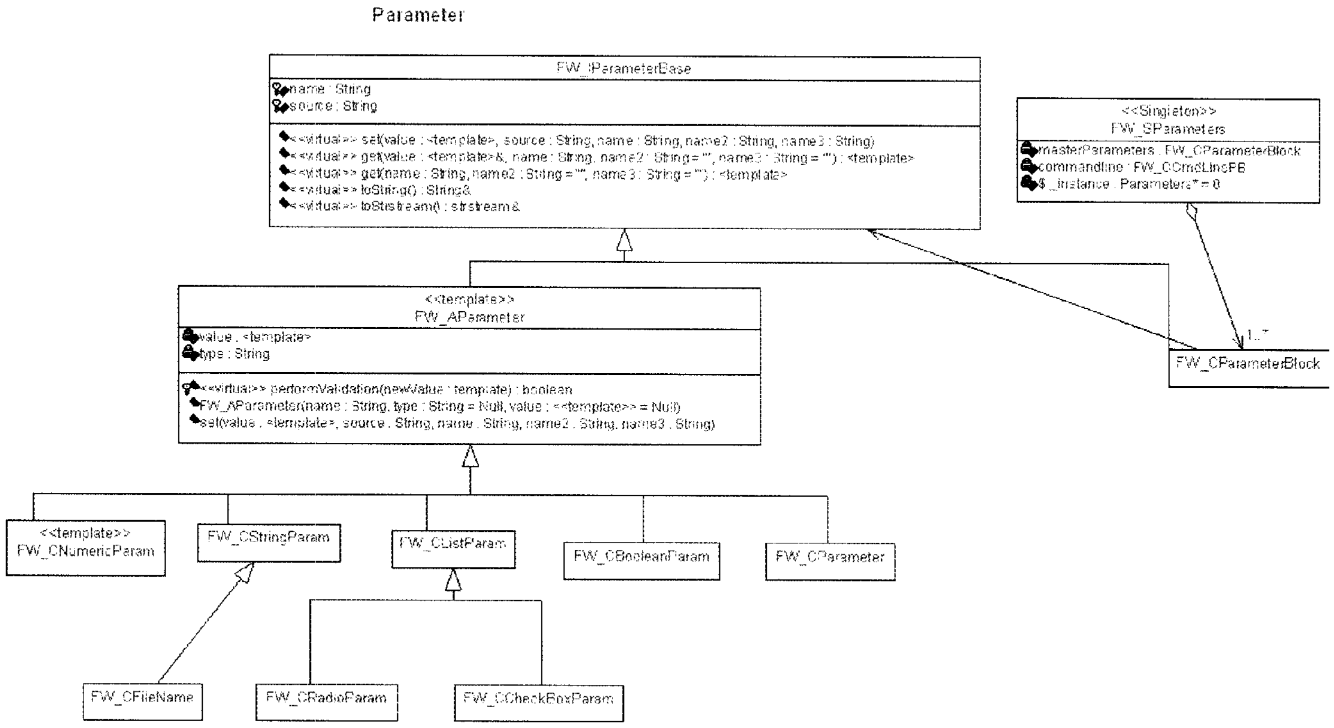

Parameter
FW_IParameterBase
name : String
source : String
<<virtual>> set(value : <template>, source : String, name : String, name2 : String, name3 : String)
<<virtual>> get(value : <template>&, name : String, name2 : String = "", name3 : String = "") : <template>
<<virtual>> get(name : String, name2 : String = "", name3 : String = "") : <template>
<<virtual>> toString() : String&
<<virtual>> toStrstream() : strstream&
<<Singleton>>
FW_SParameters
masterParameters : FW_CParameterBlock
commandline : FW_CCmdLinePB
$_instance : Parameters* = 0
1..*
FW_CParameterBlock
<<template>>
FW_AParameter
value : <template>
type : String
<<virtual>> performValidation(newValue : template) : boolean
FW_AParameter(name : String, type : String = Null, value : <<template>> = Null)
set(value : <template>, source : String, name : String, name2 : String, name3 : String)
<<template>>
FW_CNumericParam
FW_CStringParam
FW_CListParam
FW_CBooleanParam
FW_CParameter
FW_CFileName
FW_CRadioParam
FW_CCheckBoxParam

Figure 6. Parameter-Handling Class Hierarchy

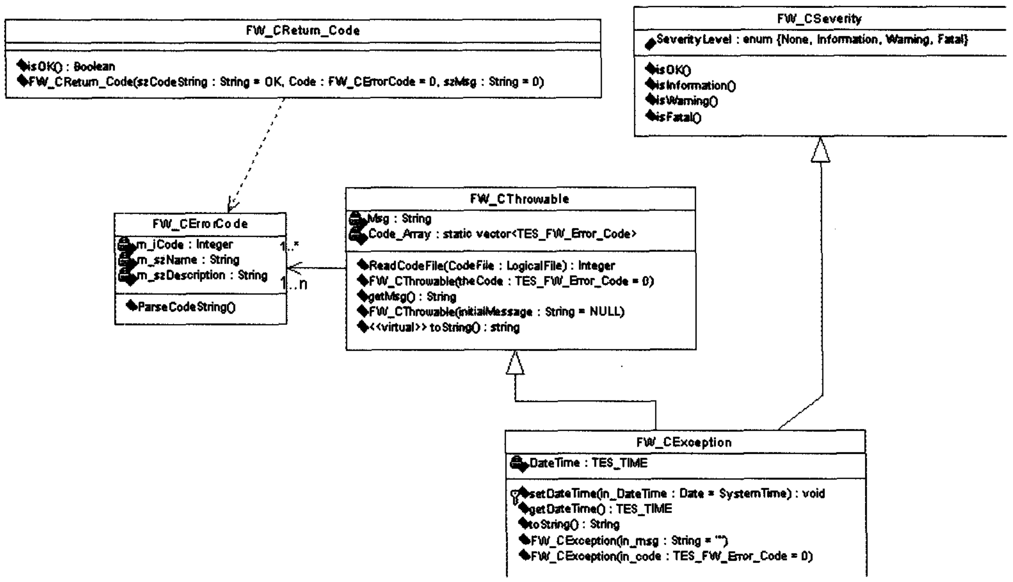

FW_CReturn_Code
isOK() : Boolean
FW_CReturn_Code(szCodeString : String = OK, Code : FW_CErrorCode = 0, szMsg : String = 0)
FW_CSeverity
SeverityLevel : enum {None, Information, Warning, Fatal}
isOK()
isInformation()
isWarning()
isFatal()
FW_CErrorCode
m_iCode : Integer
m_szName : String
m_szDescription : String
ParseCodeString()
1..*
1..n
FW_CThrowable
Msg : String
Code_Array : static vector<TES_FW_Error_Code>
ReadCodeFile(CodeFile : LogicalFile) : Integer
FW_CThrowable(theCode : TES_FW_Error_Code = 0)
getMsg() : String
FW_CThrowable(initialMessage : String = NULL)
<<virtual>> toString() : string
FW_CException
DateTime : TES_TIME
setDateTime(in_DateTime : Date = SystemTime) : void
getDateTime() : TES_TIME
toString() : String
FW_CException(in_msg : String = "")
FW_CException(in_code : TES_FW_Error_Code = 0)

Figure 7. Exception-Handling Hierarchy

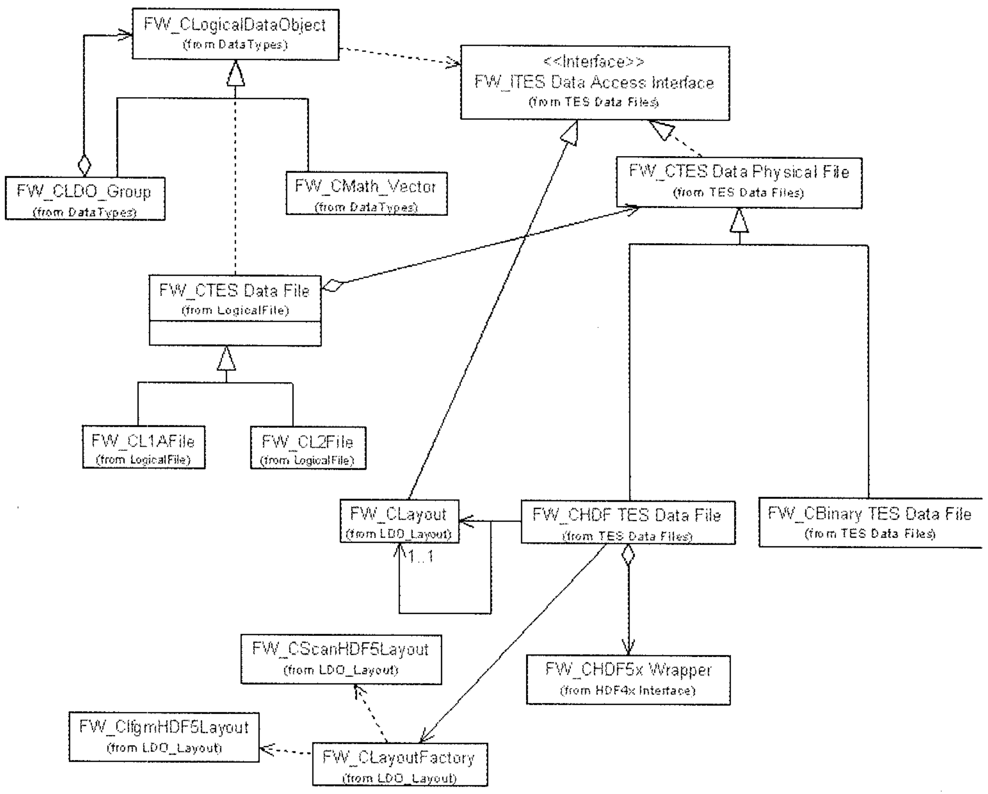


Figure 8. Logical and Physical File Relationships

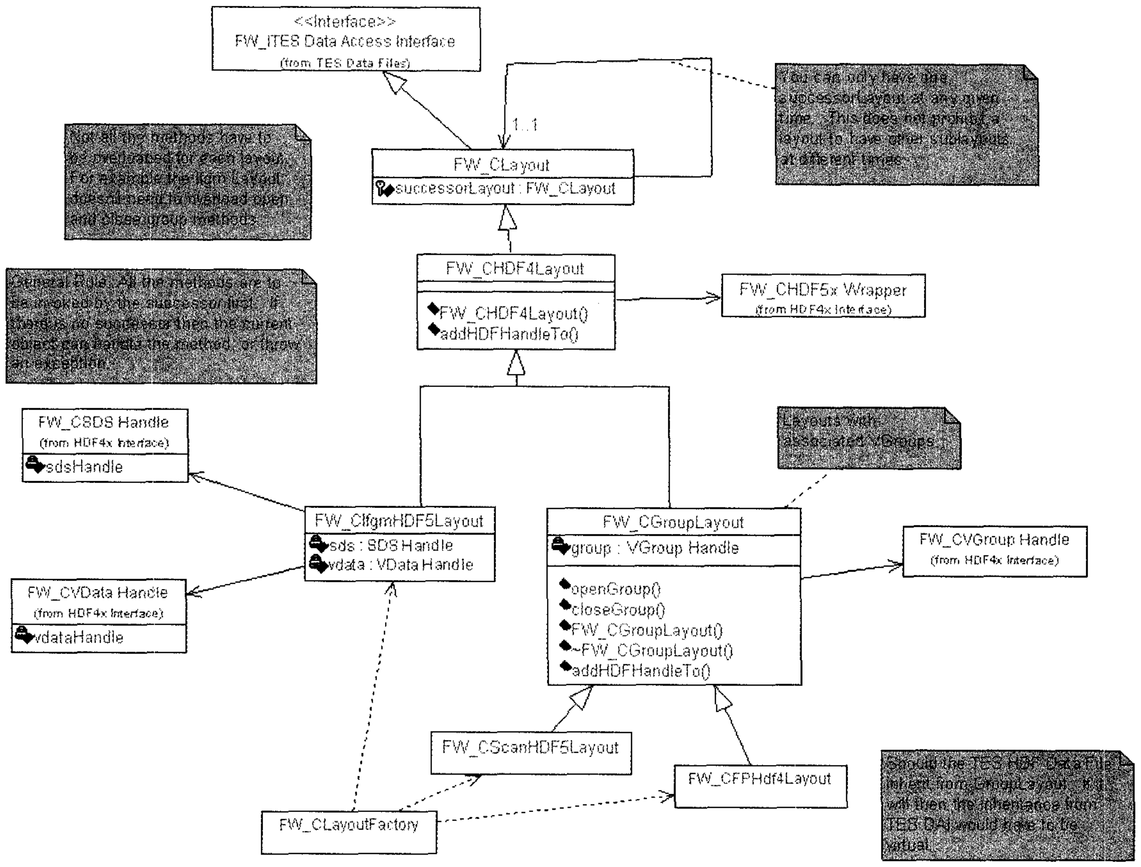

<<Interface>>
FW_ITES Data Access Interface
(from TES Data Files)
1..1
You can only have one successorLayout at any given time. This does not prohibit a layout to have other sublayouts at different times.
Not all the methods have to be overloaded for each layout. For example the Igm Layout doesn't need to overload open and close group methods.
FW_CLayout
successorLayout : FW_CLayout
FW_CHDF4Layout
FW_CHDF4Layout()
addHDFHandleTo()
FW_CHDF5x Wrapper
(from HDF4x Interface)
General Rule: All the methods are to be invoked by the successor first. If there is no successor then the current object can handle the method, or throw an exception.
FW_CSDS Handle
(from HDF4x Interface)
sdsHandle
Layouts with associated VGroups
FW_CIfgmHDF5Layout
sds : SDS Handle
vdata : VData Handle
FW_CGroupLayout
group : VGroup Handle
openGroup()
closeGroup()
FW_CGroupLayout()
~FW_CGroupLayout()
addHDFHandleTo()
FW_CVGroup Handle
(from HDF4x Interface)
FW_CVData Handle
(from HDF4x Interface)
vdataHandle
FW_CScanHDF5Layout
FW_CFPHdf4Layout
Should the TES HDF Data File inherit from GroupLayout. If it will then the inheritance from TES DAI would have to be virtual.
FW_CLayoutFactory

Figure 9. Layout Class Hierarchy