AFRL-VS-PS-
TR-1998-1070

UNIFORM INTERFACE FOR MULTIPLE SATELLITE SYSTEMS STUDY

Gloria Connor
John Forrest Harrell

GenCorp Aerojet
P. O. Box 296
1100 West Hollyvale Street
Azusa, CA 91702

August 1998

Final Report

AIR FORCE RESEARCH LABORATORY
Space Vehicles Directorate
3550 Aberdeen Ave SE
AIR FORCE MATERIEL COMMAND
KIRTLAND AIR FORCE BASE, NM 87117-5776

AFRL-VS-PS-TR-1998-1070

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data, does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report has been reviewed by the Public Affairs Office and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nationals.

If you change your address, wish to be removed from this mailing list, or your organization no longer employs the addressee, please notify AFRL/VS, 3550 Aberdeen Ave SE, Kirtland AFB, NM 87117-5776.

Do not return copies of this report unless contractual obligations or notice on a specific document requires its return.

This report has been approved for publication.

GEORGE S. SCHNEIDERMAN
Project Manager

FOR THE COMMANDER

KEITH SHROCK, D-III
Acting Chief, Space Sensing and
Vehicle Control Branch

CHRISTINE ANDERSON, SES, USAF
Director, Space Vehicles Directorate (VS)

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	27 August 1998	Final Report - 8 May 1997 through 4 June 1998

4. TITLE AND SUBTITLE
Uniform Interface for Multiple Satellite Systems Study

5. FUNDING NUMBERS
F29601-97-C-0051
PE: 63401F
PR: 2181
TA: TC
WU: 02

6. AUTHOR(S)
Gloria Connor
John Forrest Harrell

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
GenCorp Aerojet
P. O. Box 296
1100 West Hollyvale Street
Azusa, CA 91702-0296

8. PERFORMING ORGANIZATION REPORT NUMBER
Report 11173

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory
Space Vehicles Directorate
AFRL - VSS-Satellite Control
3550 Aberdeen Ave SE
Kirtland Air Force Base, NM 87117-5776

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
AFRL-VS-PS-TR-1998-1070

11. SUPPLEMENTARY NOTES
None

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 Words)*

This report attempts to model a multi-satellite ground telemetry processing system (MSG) for reducing the level of effort associated with extending a system to supporting new satellites. The use of object-oriented (OO) hierarchy and abstraction alone does not sufficiently reduce the inherent complexity of a MSG due to the number and diversity of classes required to support such a system. This approach reduces the potential for multiple branches and layers of subclasses without losing generality and thus maximizes code reuse. The model was developed using ordinary, well documented OO patterns combined with a novel approach for retrieving class attributes external to the source code. A database management system with database and query language capabilities is used for retrieving attributes during system initialization. This allows chaining classes together for building a processing structure without hard-coding anything more than the abstract classes being used. A simplified version of the model was tested in the 'proof-of-concept' and shown to work correctly. In this test, it was possible to support a new satellite just by inserting attribute information into the database without any recompilation of the code.

14. SUBJECT TERMS
Object-oriented design, satellite ground control, satellite status, relational databases

15. NUMBER OF PAGES
46

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unlimited

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

TABLE OF CONTENTS

FIGURES

Introduction

1.1 Objectives and Scope

The objectives of this report are to document an object oriented model developed at Aerojet for a multiple satellite system. The model was developed for the purpose of simplifying the process of extending the Multi-mission Advanced Ground Intelligent Control (MAGIC) System software to new applications and for supporting new satellites. The model includes an analysis of the domain, a design for multiple satellites, a framework for expanding MAGIC's functionality and analysis of extensibility based on Aerojet's analysis of the domain.

The study of the domain leading to the development of a model, is limited to health monitor processing of Electrical Power Distribution Subsystems (EPDS) and Attitude and Control Subsystems (ACS) for the Defense Support Program (DSP) and (UFO) satellites rather than the entire domain of MAGIC. The model itself is developed only for ground telemetry processing and avoids the area of middleware, which is covered in a separate report titled "Controller and Communications Middleware Survey and Evaluation For the Next-Generation, Common Satellite Ground Station".

Note even though the MAGIC system no longer exists the objectives are still the same.

1.2 Notation

Universal Modeling Language (UML) notation is used for illustrating the analysis and design described in this report. See Figure 1 for the symbols used in this report.

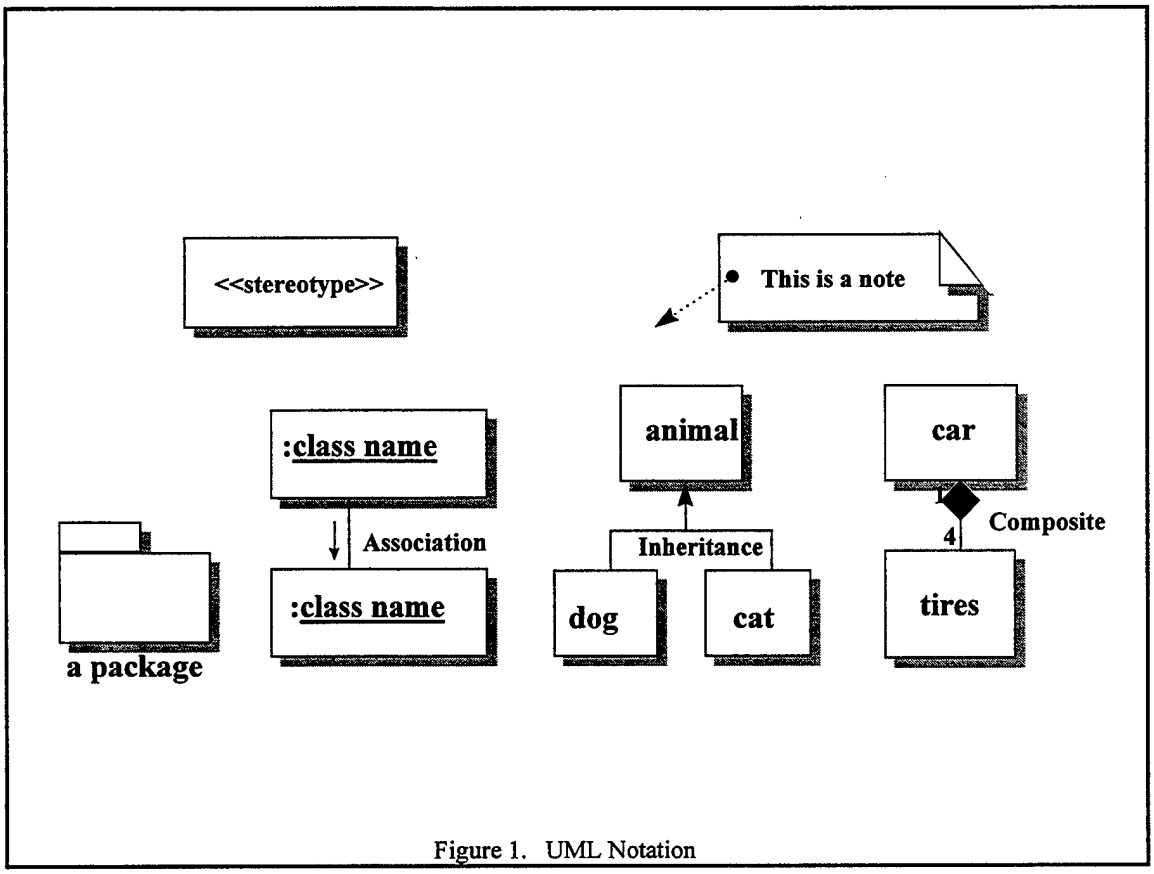


Figure 1. UML Notation

1.3 Approach Used For Developing Model

1.3.1 Overview

The basic approach decided upon for developing a model for the MAGIC domain is to generalize and parameterize classes, where applicable, to avoid a large number of subclasses and use a DBMS to store and retrieve class attributes.

To achieve generalization, subclass behavior is designed into a class's member variables or attributes rather than modifying or adding member functions through subclassing. Each set of attribute values that defines some specific behavior is stored in the database during the software development phase and retrieved by the software at runtime.

In a relational database, a set of values may be stored in a row of a table whose fields map back to the member variables of a class. In an object oriented database, objects could be set for specific behavior, stored in the database and used during runtime for replicating that behavior. In either case, an index may be used to represent a specific behavior for a class which can be used to query the database for retrieving the row or object.

An instantiation of a class and a set of attributes describing subclass-like behavior is referred to in this report as an 'object instance'.

1.3.2 Generalize Classes

The purpose of generalizing is to reduce the effort of writing new specialized code, improve reuse and in the author's opinion, reduce the overall volume of code. If complexity is not introduced through generalization, a smaller volume of code should reduce the amount of time spent on software development, testing, and later, maintenance. The tradeoff to generalization is, to some degree, a loss of flexibility. This means the analysis of the domain must be carefully made or flexibility built in through another means.

An example of how generality is applied to the design described later in this report, is where a potentially huge set of highly specialized channel extraction subclasses were avoided in favor of one generic and relatively simple channel extraction class.

1.3.2.1 Strategy and Bridge Design Patterns

The "Strategy" and "Bridge" design patterns are used to hide implementation from within the generalized class. These patterns move specific implementation from the subclass and member functions, into another class hierarchy which can be used polymorphically by the generalized class.

1.3.2.2 Run Time Type Information (RTTI)

Classes and subclasses, used by generalized classes in the "Bridge" or "Strategy" design pattern, may be represented as integers and assigned to member variables so that the subclass is completely hidden from the generalized class.

Recent versions of C++ compiler provide run-time type identification (RTTI) and support for RTTI. These new features support converting data types and class types into integers and back again, which simplifies and improves our model. However, RTTI values change from one compilation to the next and therefore must not be saved in the database without updating the database after every compilation. A small code change to avoid updating the database, is possible but not described in detail here. Since RTTI was not available for developing the 'proof of concept' demonstration, integer values were simply hard coded and a switch statement used for instantiating the correct class.

1.3.2.3 Factory Design Pattern

A modified version of the "Factory" design pattern is used to instantiate and initialize classes whose types are represented by integers and return an abstract pointer of the new object back to the class calling Factory where it may be used polymorphically.

This approach lends itself to completely hiding subclasses and future subclasses from the classes using the Factory for this purpose. Figure 2 illustrates how Factory is used by the proposed framework for instantiating classes

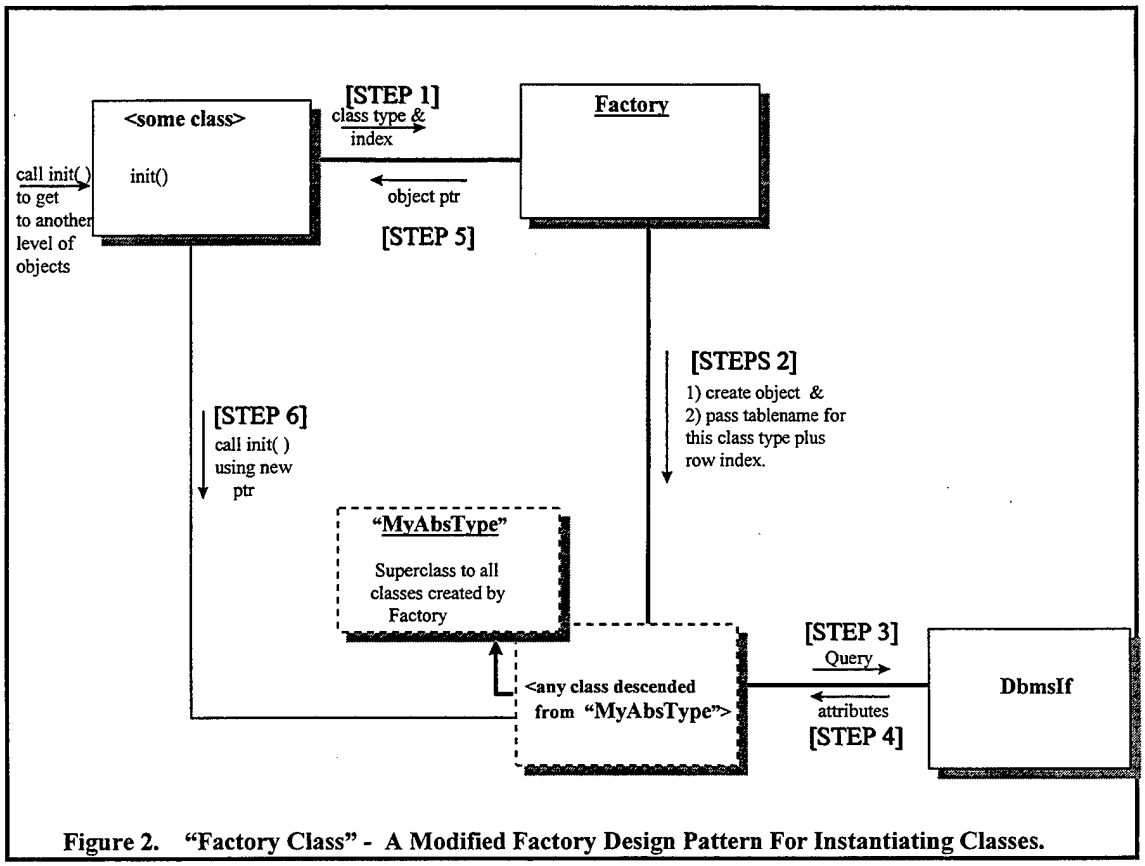


Figure 2. "Factory Class" - A Modified Factory Design Pattern For Instantiating Classes.

1.3.2.4 Common Abstract Parent Class

All but a few classes in the proposed framework are derived from a common abstract class to simplify the design in the "Factory" class.

1.3.2.5 Store Attributes In Database

Each set of values required to define an 'object instance', is stored in a database, outside the class code to promote a loosely coupled framework.

The classes where this may be applied are (a) extraction class types for storing information such as bit length, and bit offset (b) display formatting classes for storing titles, possibly screen layout, channel label strings, etc. and (c) processing classes for storing alarm limits, and floating point coefficients.

1.3.2.6 "Chain of Responsibility" Design Pattern

Processing is decomposed into small steps such that the input stream may be loosely coupled from the final display formatting step to promote better reuse of the classes responsible for processing the data. This approach is similar in concept only, to a design pattern known as "chain of responsibility".

1.3.3 DBMS Query Language

A DBMS query language is used to simplify both the design and development of the system by providing the software with the ability to retrieve attributes from the database for assigning behavior. As each set of attributes are retrieved, additional layers of classes are revealed and in turn, instantiated and initialized with attributes, so the query provides a mechanism for building a chain of processing steps. Sometimes information about the input data must be joined with other information for querying and building the model's structure.

The advantage is that it moves much of the work of extending a system, from software development/maintenance to SQL or OQL. This approach should reduce the time and simplify the effort of extending systems for new satellites. It is conceivable that similar satellites added to the system, may only require changes to data in the database and not even require recompilation of the software.

Another advantage is the amount of coding necessary for linking lower level processing steps to an architecture is reduced because the processing steps are in a sense, recovered from the database through chaining attributes.

Another unexpected advantage is it simplifies operator setup since the operator only needs to select an input source and the output (E.G., Display format instance) and doesn't need to know about the parameters or processing steps for processing the data. The software and database will determine, for example, which set of alarm limits or floating point coefficients to use or how the data processing is to be customized for a particular instance of display format.

An advantage to using an established SQL or OQL (through DBMS APIs) to query the database, is that the developers do not need to design and develop a new method for accessing behavior attributes from say, a file or s/w table.

Integrity constraints, provided by some DBMS, may be used for validating SQL insert statements entered during development and automatically add, delete, or modify rows in tables as other rows are added, deleted, or modified in other tables, by the user.

The disadvantage of a DBMS could be an apparent loss of performance during real-time or near-real-time processing as the software waits for a reply to a query. The model in this report performs all querying during an initialization phase, before any processing occurs. In practice, this will require an operator to allow sufficient time for system initialization before real-time satellite support begins.

Note, a relational DBMS is used throughout this report for illustrating the feasibility of the model. It is not the intention of this report to suggest an ODBMS is less suitable than an RDBMS.

1.4 Assumptions

1.4.1 Data Not stored in Object Oriented Format

The design is based on an assumption that data is not stored in an object oriented format. The design could easily be modified to allow for this but wasn't done here because it requires some thought about reprocessing data formatted with older versions of object oriented data.

1.4.2 One Stream Input

For the sake of simplifying the discussion of Back End processing, the model in this report assumes there is only one stream or data type per process, for the most part. To handle multiple streams, small changes are needed to the model.

1.4.3 One DEAD For Each DMAD

For the same reason as above, the design appears to assume there is one DEAD for each DMAD. Actually, the design doesn't prohibit multiple DMAD for each DEAD but it does bring up several questions, one of which is the impact on performance on other supported DMADs while handling new requests and handling contradictory requests. In practice though, one DEAD for each DMAD may be impractical and inefficient.

1.5 Conventions

With few exceptions, the naming convention used for subclasses in this report and in the Proof-of-Concept is for the subclass to append a name string to the end of the parent class name.

Model

2.1 Analysis

Figure 3 illustrates the major subsystems and their interfaces used for describing the proposed analysis and design.

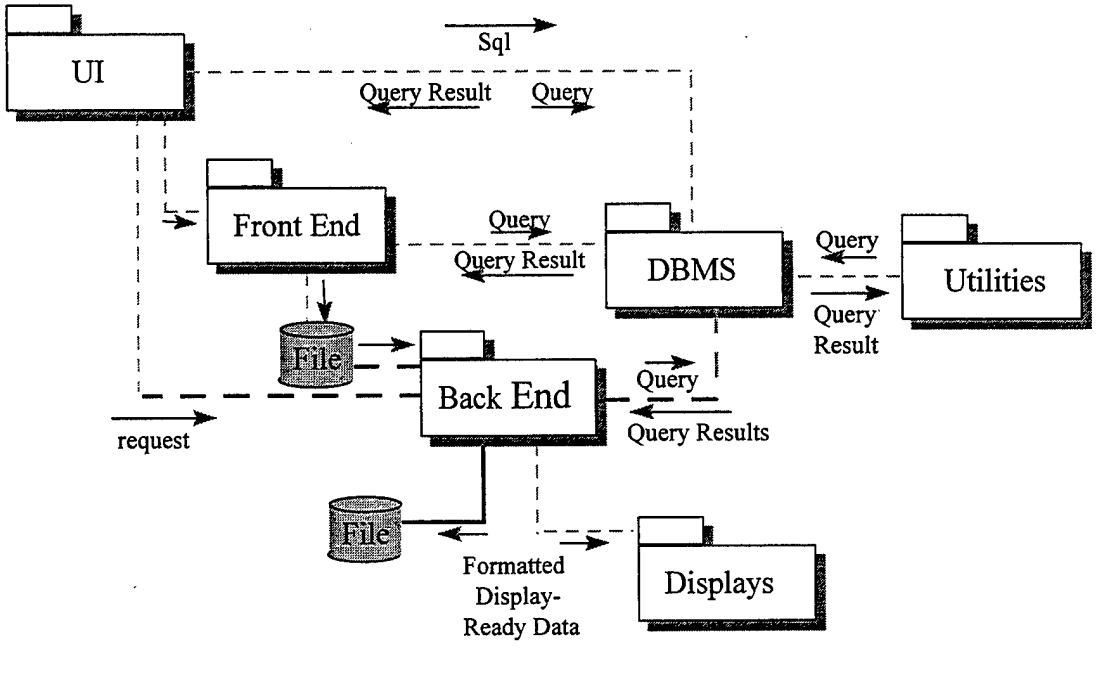


Figure 3. System Interfaces

2.1.1 Back End

Back End processing generally consists of near real-time health and sensor monitoring and non-real-time modeling after the stream data has been preprocessed (i.e., frame-synchronized, decoded, decrypted, time-tagged, etc.) by the Front End.

Figure 4 illustrates Back End processing and major external interfaces.

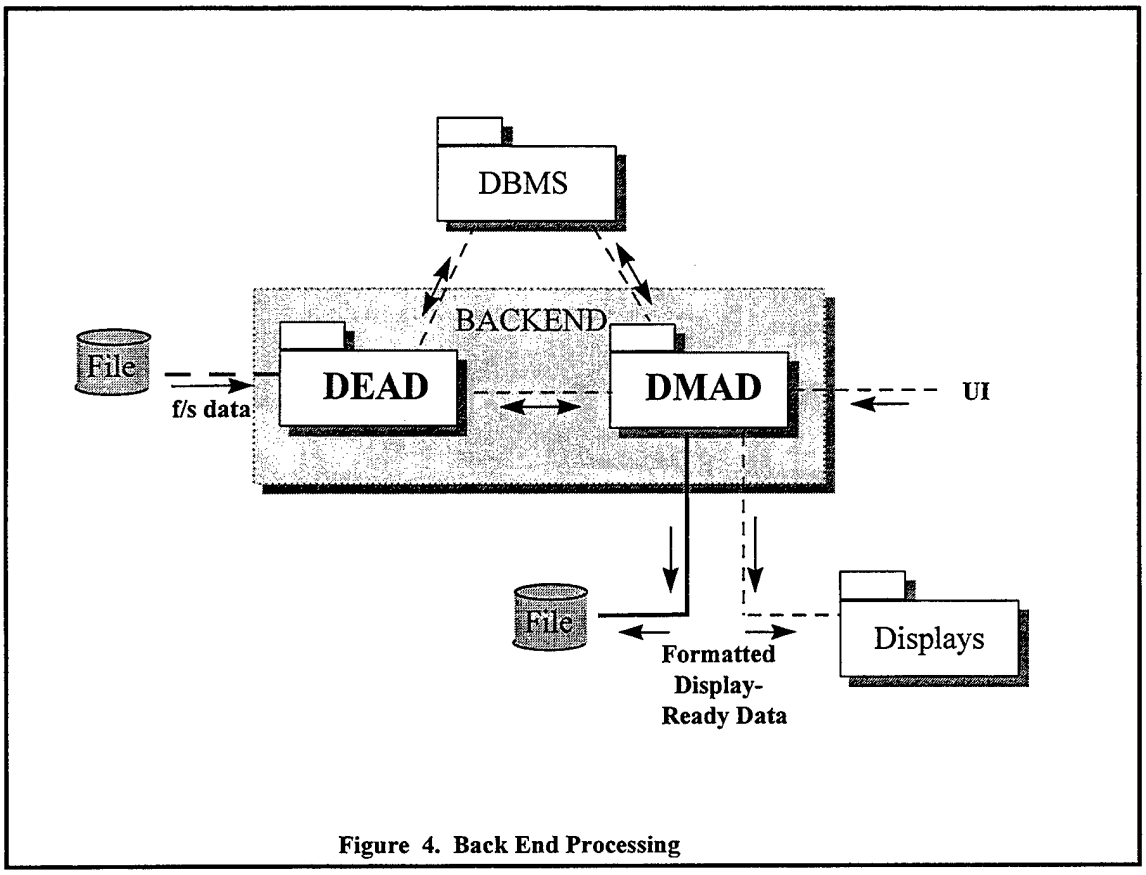


Figure 4. Back End Processing

The first of two major parts in the Back End will be called the Data Extraction and Distribution (DEAD). This process is responsible for decommutating channels, extracting frames, and sending channel and frame data to the second part of the Back End which is Data Modeling and Display Formatter (DMAD).

The DMAD, is responsible for processing telemetry data and sending formatted output to an external display process or a file. Processing in the DMAD includes floating point conversion, alarm checking, formatting data for display, etc.

2.1.2 Displays

The displays could be a collection of display processes used for displaying health and sensor data that were previously processed and formatted by the Back End.

No attempt was made to seriously model a display in this report since it is preferable to use a COTS package or one of the commercially available display building tools, such as Visual C++ with MFC. Some of these tools are mainly platform dependent, come in their own framework and use design patterns which make it difficult to port classes from one set of tools to another (i.e., from one platform to another).

2.1.3 Front End

Although the Front End was not originally included in the original domain analysis, it would appear that the approach described for Back End, is applicable to the Front End for the same reasons and benefits as the Back End. That is, we should be able to generalize subclasses through storing attributes, such as sync code, frame length, decoding method, in the database. On the other hand, it is not fully understood how well the design will perform during real-time processing because the front end may be less predictable and may require information from the database for continued processing which may impact performance. For example, most satellites can switch to an emergency mode without notice but the framework presented in this report can only accommodate the requested input data types. There are a few possible strategies for handling changes, such as a format change, in real-time without making major changes to the proposed but some additional analysis is required.

2.1.4 Database

2.1.4.1 Schema

The schema required to support this model in a relational database consists of roughly one table for every class whose columns or fields correspond to the member variables of a class and a row to an 'object instance'. In general, the key to a row is an index which may be stored in the attributes of another row of another table (or class) to allow the key holder to create an instance of the first class. For example, if an instance of a channel class requires floating point conversion by the polynomial method, the channel class table will name the polynomial method class type (integer) in one of its fields and the row index (integer) in another field, for pointing to the correct set of coefficients within the polynomial method class table.

Other information that may be useful for storing in the database but not covered elsewhere:

(i) Scheduling information for the purpose of automating processing in a multi-satellite system
(ii) Metrics for costing by class name or hierarchy, number of interfaces, etc.
(iii) Tables about the users for supporting authorization. For example, the user interface may first get authentication from the user and then use the user name for looking up authorization on that user to see if he/she is allowed to operate the system for a particular satellite or view one type of display and not another. Note that Oracle, and maybe others, have features for individual and group ownership of data (in the

database). This is good only for viewing, adding and deleting data from the tables in the database and doesn't appy to the processes outside the database.

2.1.4.2 Utilities

Database utilities are the tools for supporting changes to the database which are normally associated with the software development and software maintenance phases, mentioned earlier. New satellites that are similar to other satellites stored in the database which don't require new tables (or classes) can be added to the 'system' through SQL without any software changes.to the Back End and hopefully, Front End as well. If user interface for the system is designed to retrieve information from the database (regardless of whether it uses the framework), then menus and options will also be updated through utilities.

Utilities may also include support for the software design phase associated with extending the system by generating rough drafts of interface specifications based on information stored about another spacecraft and automatically changing details according to parameters entered by the user. After the rough draft is created, additional changes are edited by hand in a text editor according to some set formatting rules defined for the utilities. Later, another utility may be used to read and parse the new interface specification for creating SQL insert statements and using the DBMS APIs for putting the new data or even new table schema into the database. If the integrity constraints are in place, some errors will be caught and the database may be rolled back.

Other ideas to consider for developing utilities:

(i) Functions for costing based on interface specification and metric tables stored in database.
(ii) Utilities for generating rough drafts for test scenario documentation and possibly the test scripts themselves.

Utilities could probably be forced to fit the proposed framework (or vice versa) but it may be much easier to develop these utilities or tools in a language such as Perl or PL/SQL (Oracle) outside the framework.

2.1.4.3 Integrity Constraints

Integrity constraints are the features for putting constraints on data added, updated or deleted from the database tables. For example, a constraint could be defined on a field to allow only values that are in a specified set of numbers. This is important because errors that would have been caught in the past by the compiler through the use of user defined types (e.g., enums) must now be caught by the DBMS through integrity constraints features.

2.1.5 User Interface (UI)

No attempt was made to seriously model the UI for some of the same reasons given in the section on Displays. That is, it is preferable to use one of the GUI building tools some of which are platform dependent, come in their own framework and use design patterns which make it difficult to port classes from one set of tools to another (I.E., from one platform to another). RogueWave claims their zApp produces platform independent GUI code through the use of zApp APIs and plug in code for calling the underlying operating system. zApp comes with Object Factory for designing the GUI objects in conjunction with the database similar to the approach used in this model. So it would appear that these software tools support our approach.

Regardless of the model, the UI may benefit from the use of a database by using it to retrieve information for building it's list of options at runtime, to avoid hard coding these. Again, this will depend on the tools selected because some tools do not support the capability to build menus from an array of options of indeterminate size, at runtime. Aside from the database, the user interface may benefit from other aspects of the approach but not to the extent that the Back End benefits.

2.2 Design

2.2.1 Back End

2.2.1.1 Overview

To understand the design of the Back End, we first need to look at the sequence diagram in the Figure 5.

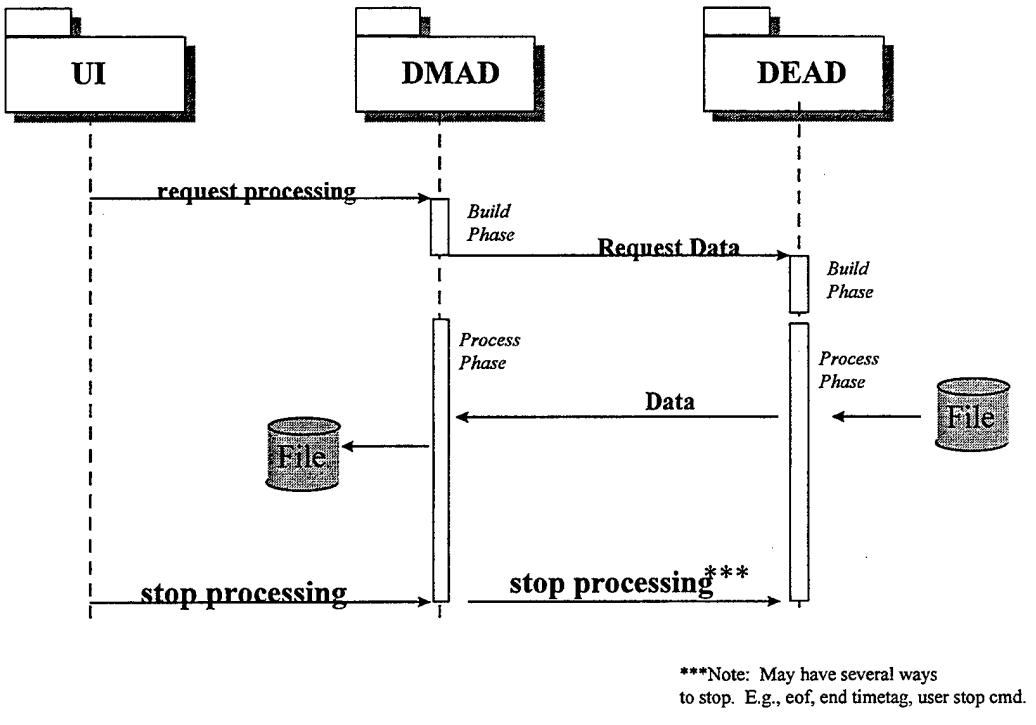


Figure 5. Sequence Diagram For The Back End

This diagram illustrates:

(i) The DMAD receives a request for processing which could be from the User Interface.
(ii) The DMAD instantiates and builds its own object structure for performing the requested processing.
(iii) The DMAD produces a list of data types needed for it's processing at the end of its build phase. The list becomes a request to the DEAD.
(iv) The DEAD receives a request from the DMAD and builds a structure for producing the requested data types while DMAD waits for input.
(v) The DEAD starts its processing phase and outputs the requested types to the DMAD.
(vi) The DMAD processes data received from the DEAD.
(vii) Eventually, processing ceases when (a) DEAD reaches EOF, and notifies DMAD or (b) user notifies DMAD to terminate, and DMAD notifies DEAD, etc.

2.2.1.2 DEAD

2.2.1.2.1 Overview

Figure 6 gives a sense of how the DEAD is structured from the abstract class types. What the abstract types in this figure do not show is how the "Flyweight" design pattern is applied and modified for this design so that a channel object instance always processes the same channel type for the entire stream so that channel classes are not instantiating and destructing as data is processed. The result of this approach may be improved performance.

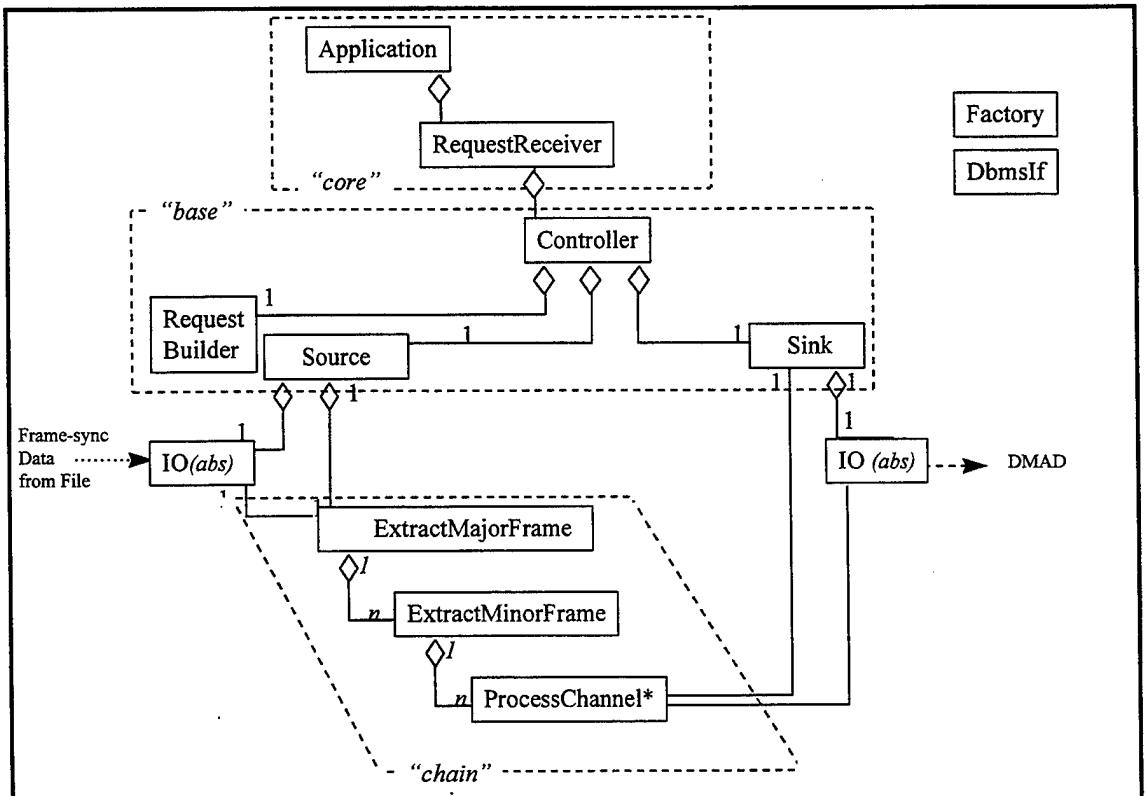


Figure 6. DEAD - Extracting and Decommutating (Class Model).

2.2.1.2.2 Build Phase

Figure 7 shows the object structure at the end of the build phase and the flow of control as the structure is being built.

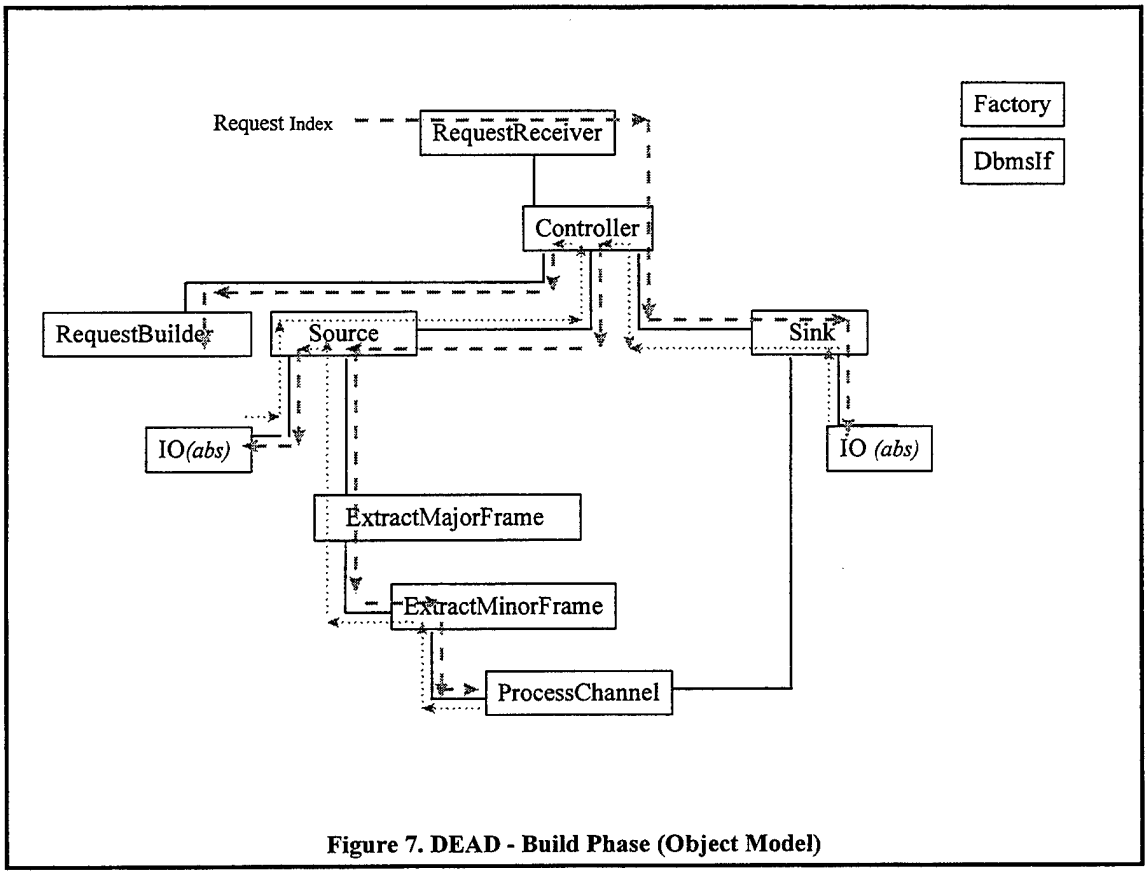


Figure 7. DEAD - Build Phase (Object Model)

Initially, the DEAD is made up of a very small core of classes. This core only has the capability to receive and process a 'request' for data from the DMAD.

After the request is received, the DEAD grows from the small "core" into a much larger structure by instantiating and initializing classes needed for processing the input data. At first, the "RequestReceiver" queries for the "Controller" type (Controller or maybe even ControllerDead) needed for this process. When the "base" classes are initialized, the request table is queried for the input data and the list of output data needed by the DMAD. The input may have information in the database about its location (E.G., node, path, file name, version, or socket description, etc.), data type (E.G., major frame types) and file format which gives the Source class enough information for instantiating the IO and class described as the input type, underneath.

The DEAD builds a "chain" of processing type classes from the input type, that are needed for extracting and decommutating the requested output types. Each time the DEAD branches, the database is queried for attributes to initialize the class member variables. During this initialization, the new object will use information from the attributes and the request to decide whether to produce another level of objects. For example, (a simplified explanation) the minor frame objects will have attributes, possibly an SQL string, for finding its channels in the

database. The channels (rows), once located, will be instantiated and owned by the minor frame object. In this example, DSP will have 128 rows in the minor frame table resulting in 128 minor frame objects. Each of these minor frames will find about 128 rows in the channel table and instantiate the ones listed in the request.

When one of the channels is reached, the frame must pass a pointer to each channel object so that classes can retrieve time tag bits or retrieve other information which will allow to retrieve the time tag later. There are at least two possible strategies for handling time tags in channels but they are not discussed here.

When the build phase is complete, the DEAD is ready to process and doesn't require further use of the database.

2.2.1.2.3 Process Phase

Figure 8 shows the object model for DEAD processing and the flow of control for extracting and decommutating channels from the data stream. Notice that data is read in at the "Source" and output at the "Sink" and the dashed lines show the direction data travels to output.

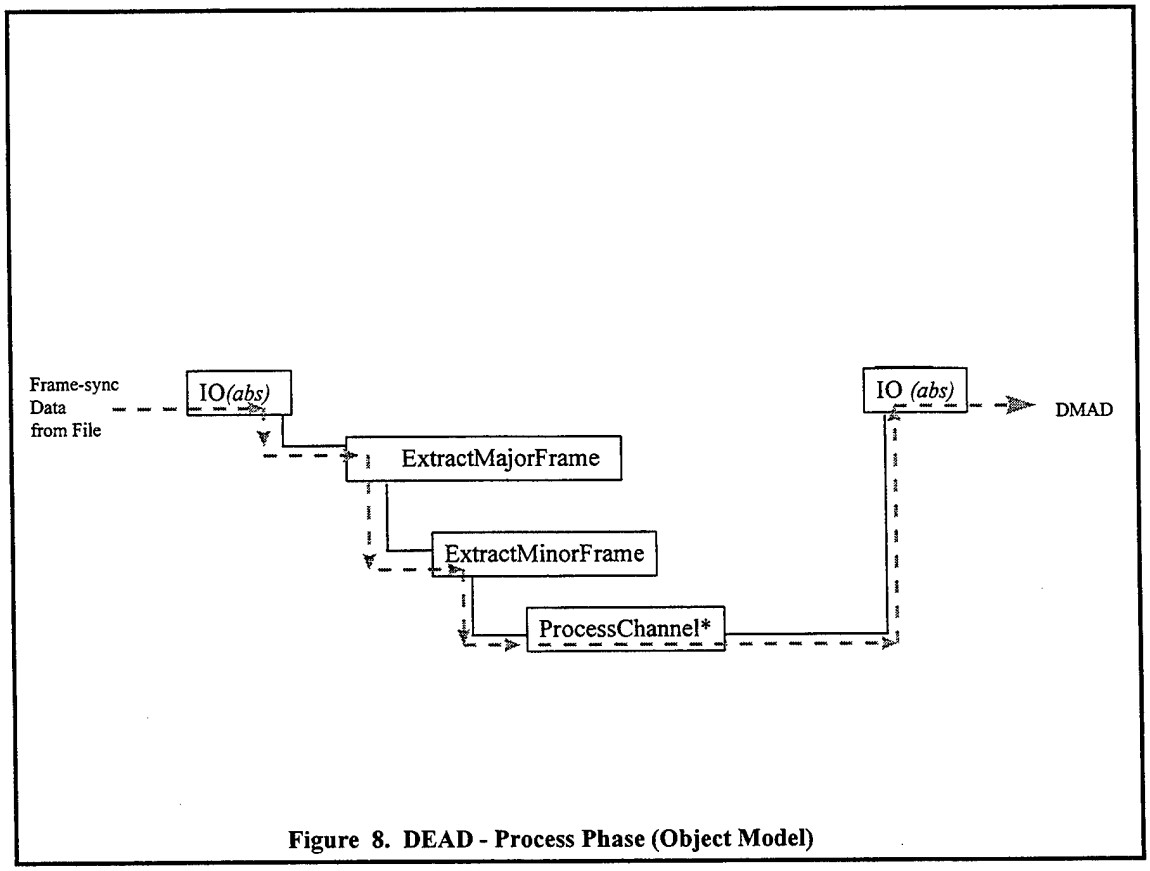


Figure 8. DEAD - Process Phase (Object Model)

In practice, it is better to pass along pointers to the data buffer and calculate new offsets rather than extracting and copying data at each level, as the class name implies, until the 'leaf' node is reached. Finally, the extracted channel and time tag data are treated simply as a sequence of bits with no other type information.

2.2.1.3 DMAD

2.2.1.3.1 Overview

Figure 9 illustrates the DMAD structure using abstract classes in the 'chain'. Figure 10 illustrates the DMAD structure for display formatting with specific subclasses named in the 'chain', mainly.

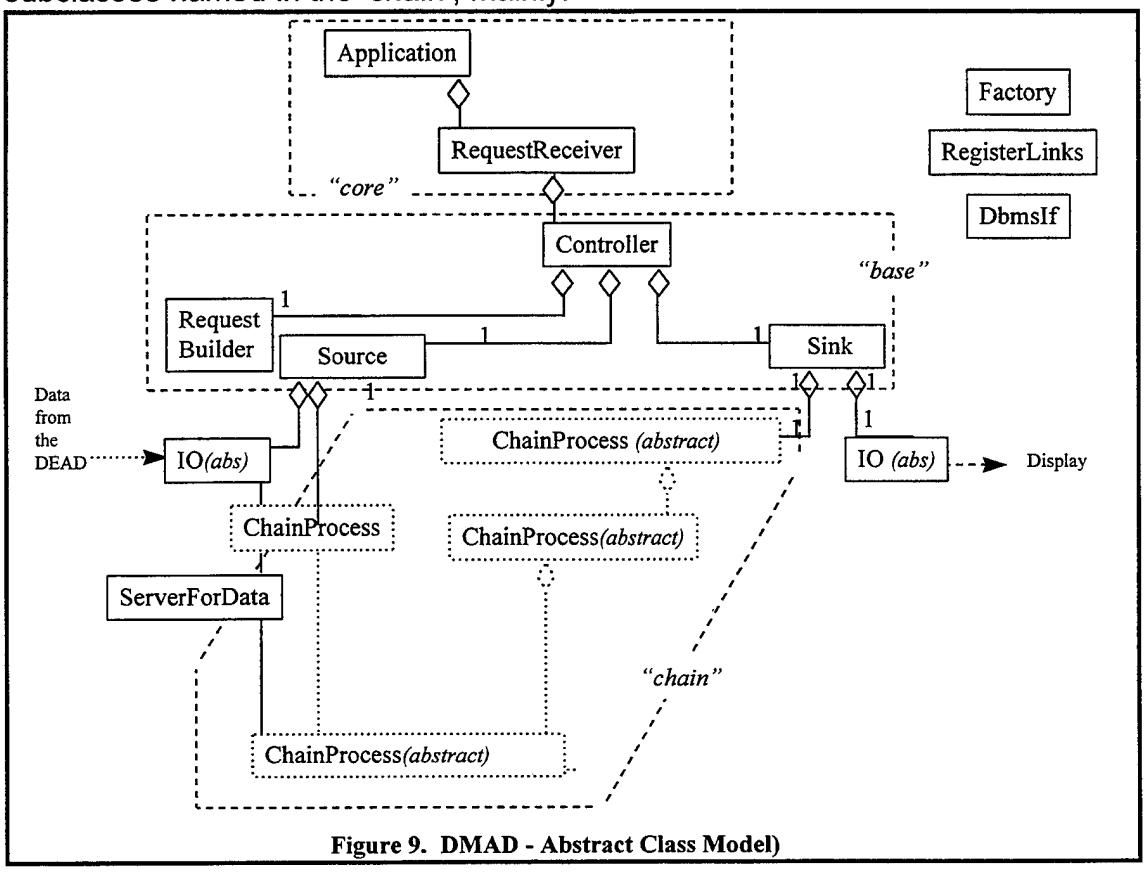


Figure 9. DMAD - Abstract Class Model)

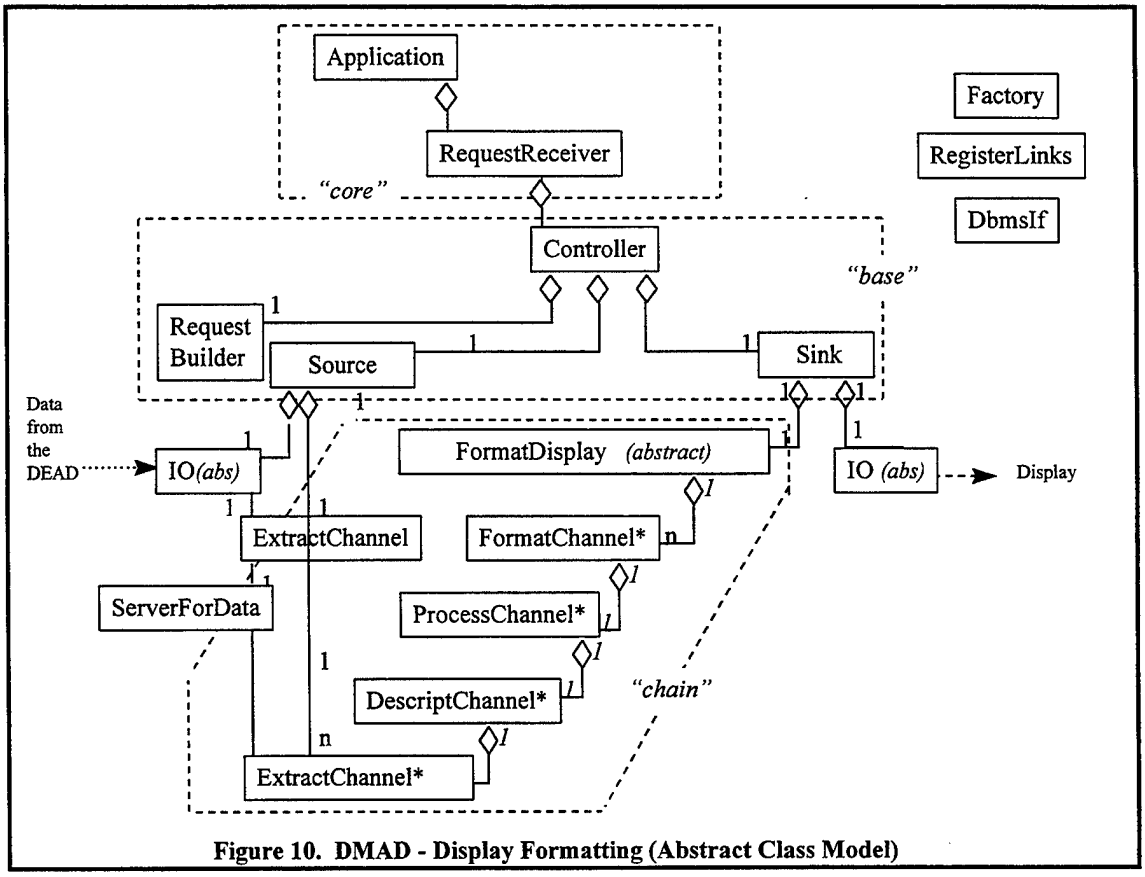


Figure 10. DMAD - Display Formatting (Abstract Class Model)

Both figures show a "core" and "base", like the DEAD, made up of RequestHandler, Controller, Source, Sink, and RequestBuilder classes. Also, like the DEAD, the DMAD has a "chain" that uses a modified version of the "Flyweight" design pattern.

The main dissimilarity between DEAD and DMAD is how the "chain" hangs. In the DMAD, the "chain" hangs from the Sink and from the DEAD it hangs from the source. With the use of features not presented in this report, it should be possible to configure the DMAD to include processing performed by the DEAD or reconfigure the two chains to break up work differently between the two processes.

From the chain, are "offshoots". In general, offshoots are subclasses in the "Strategy" or "Bridge" design pattern for doing work on behalf of the class using them. Without these offshoots, the owner class would require subclassing and lose generalization. Without generalization, it becomes more difficult for the architect to design the chain into the database for processing the data. Figure 11 shows the 'chain' segment and examples of 'offshoot' classes for formatting data for a display. Figure 12 shows the same 'chain' segment for formatting a *list* display.

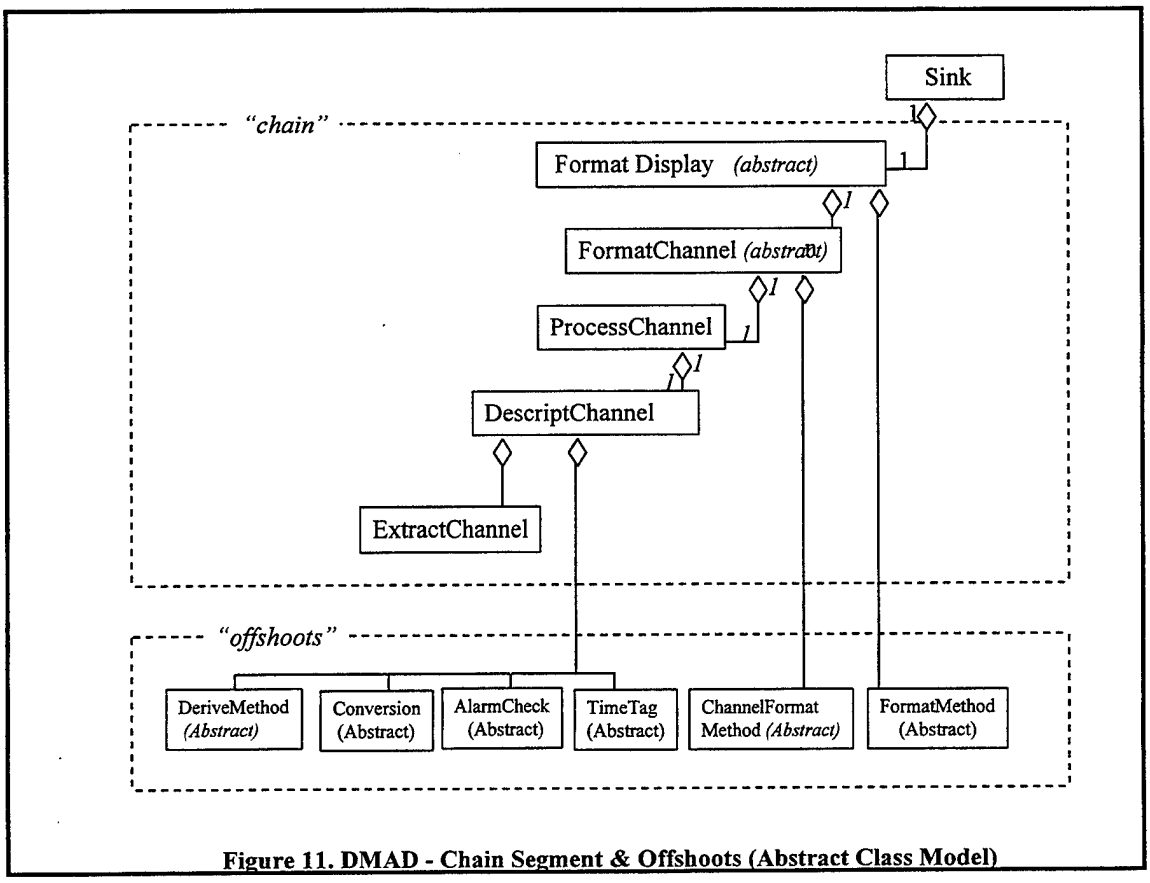


Figure 11. DMAD - Chain Segment & Offshoots (Abstract Class Model)

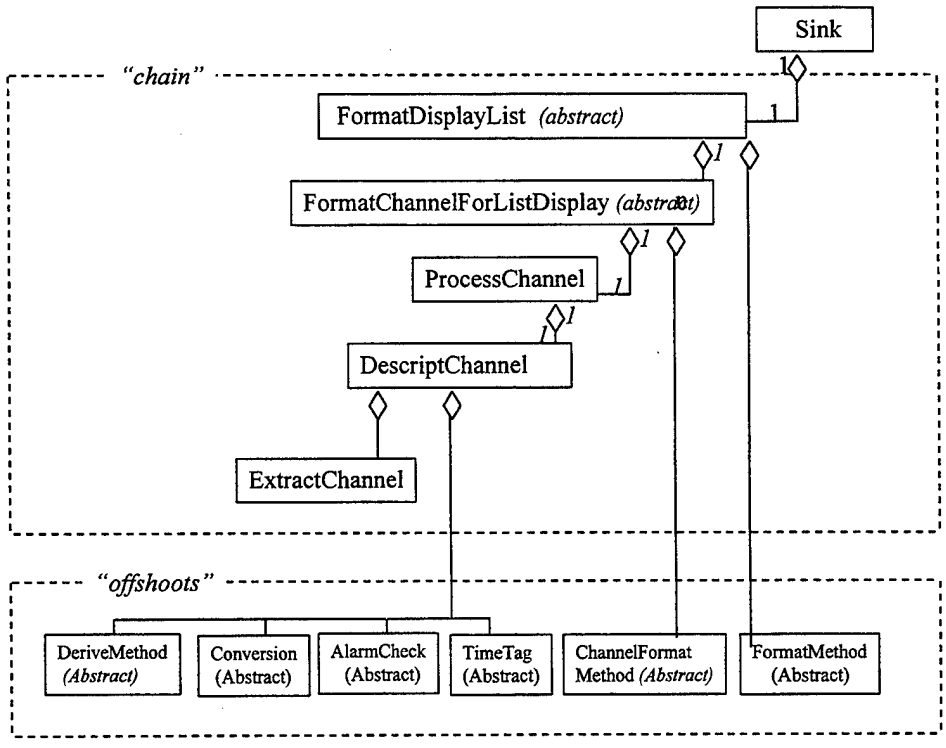


Figure 12. DMAD Chain Segment & Offshoots - For List Display Formatting (Class Model)

2.2.1.3.2 Build Phase

Figure 13 shows the order and depth of building for display formatting given the input data consists of specific major frames (on the DEAD input) and the display requires a number of generic channel types (e.g., subsystem "bus current").

Like the DEAD, the DMAD starts with a very small "core" for receiving and processing a 'request' from the user interface. The 'request' specifies an instance of a Display (IE display class type and row index) and the input data to be processed by the DEAD. With this information, the DMAD builds a "base" and a "chain" of classes using the database to fill the attributes of one class and using the factory to build the next.

The "chain" is built in much the same way as DEAD except that growth starts at the Sink class from the requested display instance (or some other ChainProcess subclass type) and list of generic channel types and builds backwards towards specific extracted or decommutated data types. See figure titled "Using DBMS for Building a Chain of Classes in DMAD".

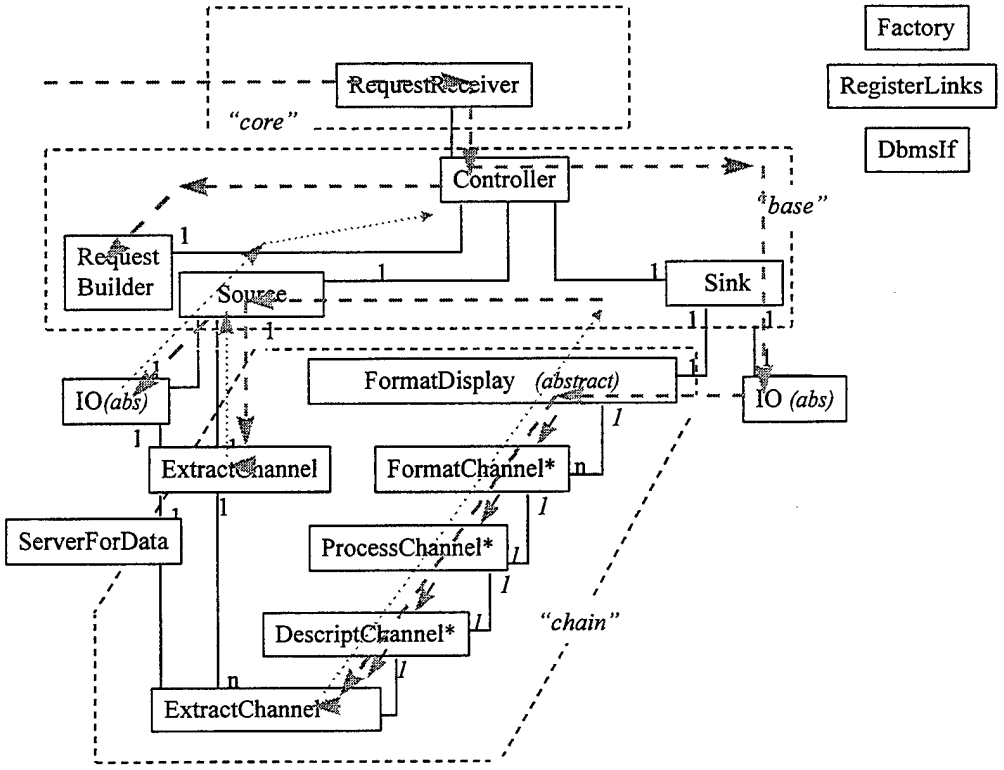


Figure 13. DMAD - Build Phase (Object Model)

The DMAD produces several small offshoots from the chain of processing classes. These offshoots are subclasses in the "Strategy" design pattern for performing floating point conversion, alarm checking, etc.

At the end of the build phase, the DMAD generates a list of extracted data types and indexes to post in the database as a request to DEAD and passes an index into the request table or 'request ID' to the DEAD. Afterwards, the DMAD is ready to process and just waits for data to arrive from the DEAD. Like the DEAD, it no longer requires the use of the database after the build phase is complete.

2.2.1.3.3 Process Phase

Figure 14 shows the direction data travels from the input at the Source to the output at the Sink for formatting a display.

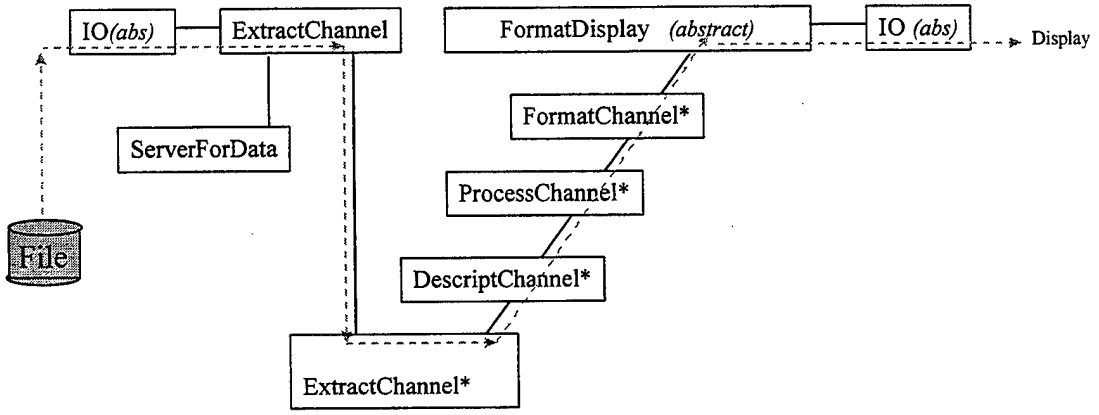


Figure 14. DMAD - Process Phase (Object Model)

2.2.2 Front End

2.2.2.1 Overview

Front End processing was not studied well enough to provide a design in this report.

We can say that the design for the Front End will include the same "core" and "base" classes introduced in the section describing Back End design. The Sink class, however, may have multiple IO(abs) for outputting data to files for different formats or simply to save bad data in a separate file. It is not known whether a "chain" should be used here but the other design patterns and approaches previously mentioned should be applicable here for meeting the objectives stated in this report.

2.2.2.2 Build Phase

The Front End has a build phase similar to the one described in Back End. However, Front End processing will most likely be performed within a single process (for a given satellite), unlike the Back End which is divided into two processes.

The UI (or possibly an automated scheduling process) will put in a request for processing a stream of raw data received from a specified satellite. This will provide the initial attributes for building the Front End and build one step at a time, just like the Back End. Ideally, the front end would be built with the capability to process all data types (all Major Frames) for a given satellite due to a certain amount of unpredictability in the stream but as stated earlier, will require more study.

2.2.2.3 Process Phase

When the data starts to process, an entry is made in the database describing the new file or socket stream, etc. in terms of satellite number and data type(s) and file-path or socket. Depending on a number of variables, this entry in the database could be updated periodically to show the time stamp of the latest data and whether the stream is still being acquired.

No additional information is available about the process phase.

2.2.3 Displays

As stated earlier, no attempt was made to model this code.

2.2.4 Database

No attempt was made to model the database utilities or schema further.

2.2.5 User Interface

No attempt was made to model the User Interface.

2.3 Framework

Figure 15 shows a skeleton application for the Front End, Back End and possibly the Displays.

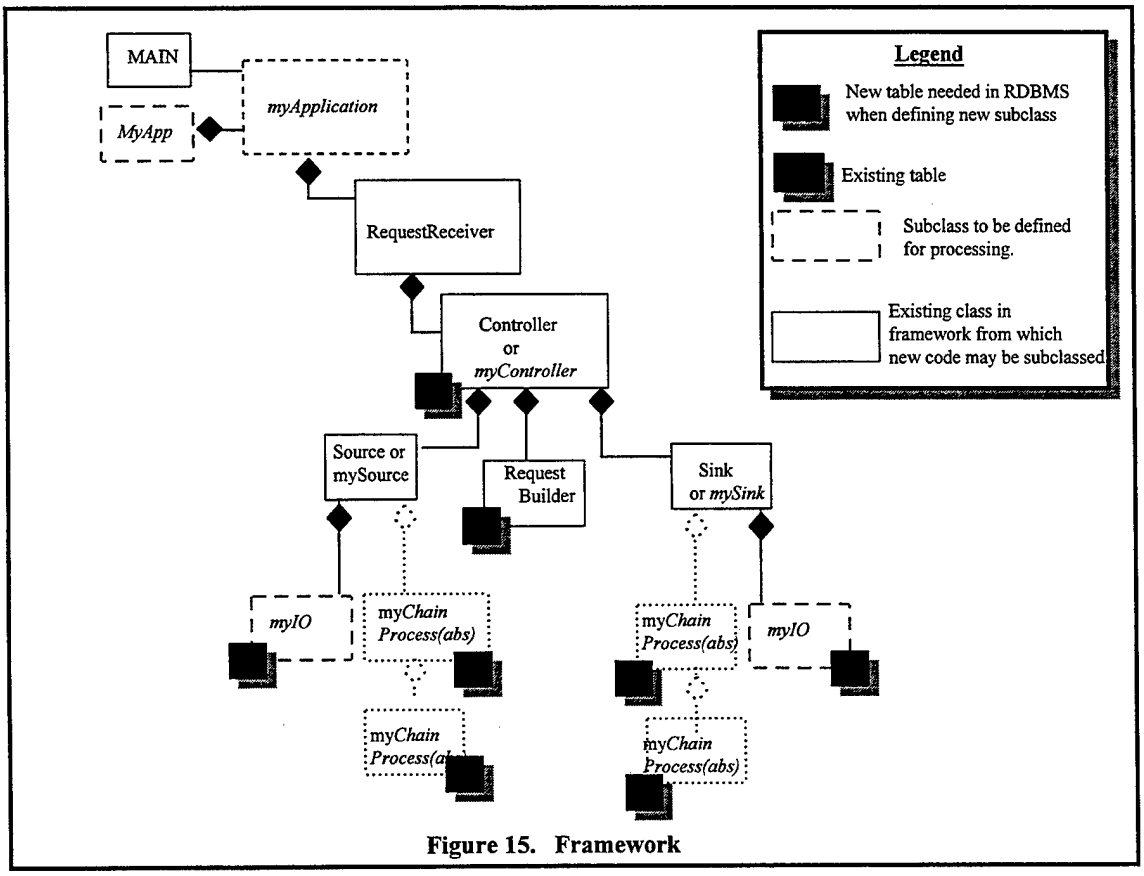


Figure 15. Framework

Figures 16 and 17 show the inheritance hierarchy of classes used in Back End processing.

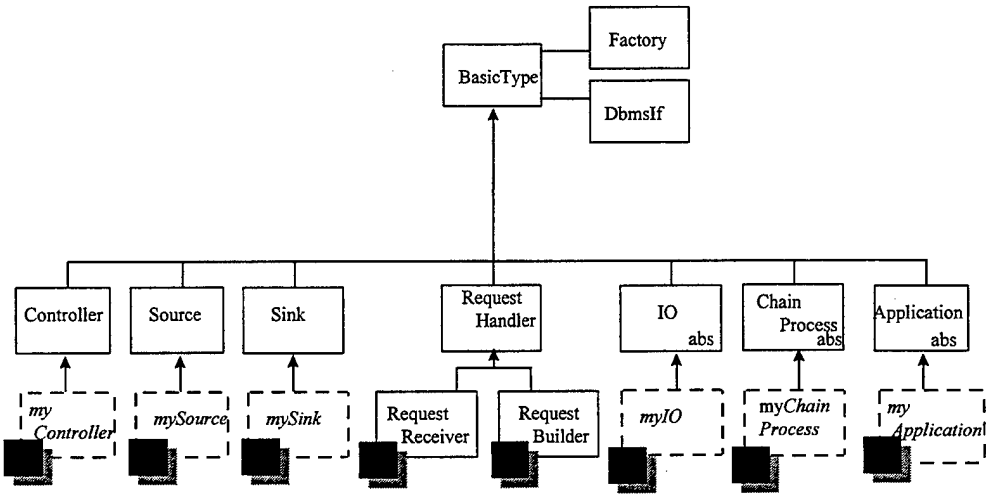


Figure 16. Framework Class Inheritance

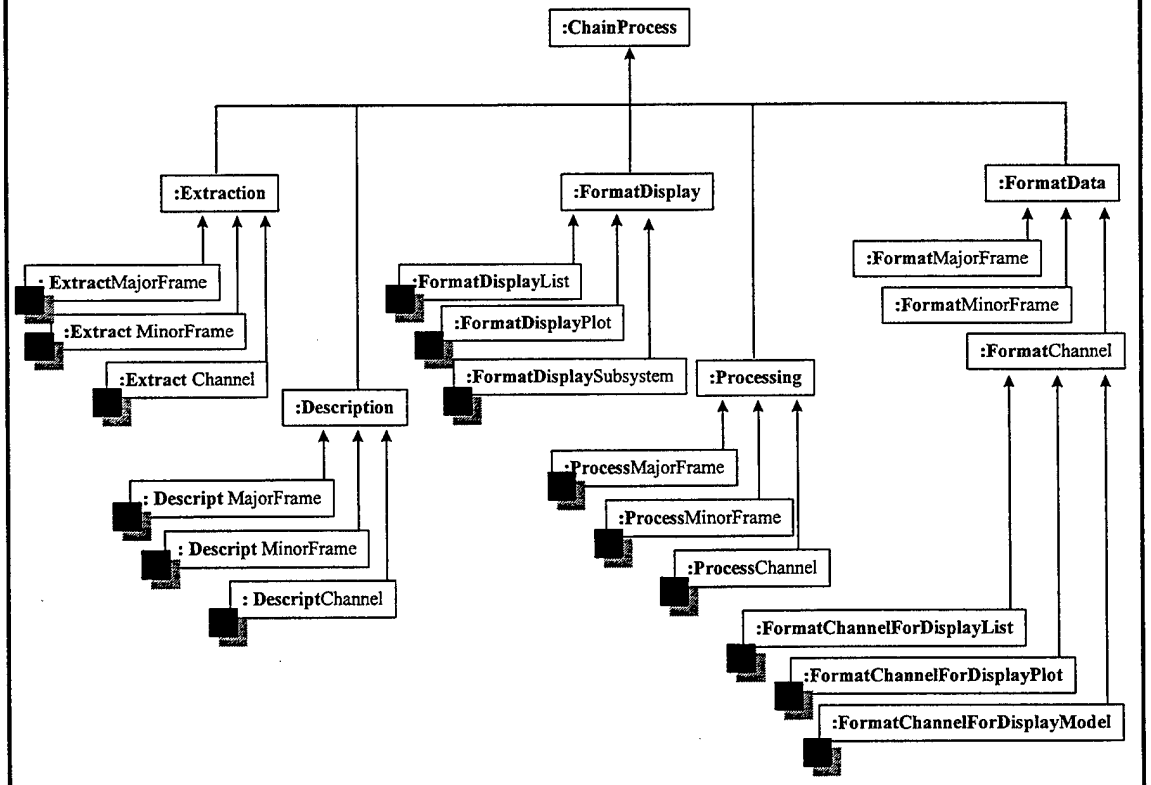


Figure 17. Class Inheritance For ChainProcess

2.3.1 Main Routine (Not a Class)

This is not a class. This is a very small generic main routine that passes arguments and control to the Application. The "Main" source code may be used in all applications derived from this framework without being recoded.

2.3.2 theApp Routine (Not a Class)

This is not a class. This is a file with one line for instantiating the "Application" subclass. Unlike "Main", every application with different "Application" subclass will require its own "theApp" to instantiate the subclass and should have a unique name to hint at which "Application" subclass is being instantiated.

2.3.3 "Application" Class

In general, "Application" handles the arguments, cleans up at the end of processing and instantiates the "RequestReceiver" subclass. Applications in the framework will most likely require "Application" subclasses only when a different set of arguments are passed in through "Main". The subclass code may be simple enough to be reused by most other applications developed from this framework.

Note that "Main", "theApp", and "Application" are presented here to promote consistency among applications developed from this framework and for reused of code. The classes, below, do not require this approach other than for starting the application so if a set of software tools provides another method of entry into the framework, there is no reason it shouldn't be used.

2.3.4 "Factory" Class

This class is a modified version of the 'Factory' design pattern. It is used in this framework for instantiating classes on behalf of other classes for hiding specific subclass types from the class using Factory. In this design, Factory passes the table name and table index to the new object so that it can retrieve it's attributes from the database. Afterwards, a pointer of the new object, casted to an abstract type is returned to the requester where polymorphism may be applied.

2.3.5 "DbmsIf" Class

This is a class for accessing a DBMS account (relational or object oriented). The class structure of this was not been developed for this report.

RoqueWave provides DBTools.h++ which may be applicable to this class.

2.3.6 "List" Class

A container like class is necessary for handling an array of "ChainProcess" objects.

RogueWave's Tools.h++ provides a class with this functionality and another class for iterating through the List.

2.3.7 "BasicType" Class

"BasicType" is an abstract class from which almost every other class in the framework is derived.

RogueWave's Tools.h++ provides an abstract class that works in the framework defined by RogueWave. To use Tools.h++, it may be necessary to subclass from RogueWaves abstract class to add the features needed for this framework.

2.3.8 "RequestHandler" Class (Abstract Class and Derived from BasicType)

This is a parent class to RequestReceiver and RequestBuilder

2.3.8.1 "RequestReceive" Class (Abstract Class and Derived from Request Class)

This class is for starting the build process after it receives a request. The 'request' could be as simple as an integer index into the database which may be used to look up the top level process class (output type) and the data source (input type) for building the structures needed for processing. In this scenario, RequestReceiver is almost superfluous but it is better to build this in than to try to add or change it later if another way to pass a request is chosen for this design.

This class may require further subclassing if other methods are used in this framework for passing requests.

2.3.8.2 "RequestBuilder" Class (Derived from Request Class)

This class is for building a 'request'. DMAD builds a list of specific channels names or data types and passes this information to the RequestBuilder to put the information in the database where it can be read by the DEAD and send an index number to the DEAD for finding that information. The User interface will put the information selected from menu by the user in the database to be accessed by the DMAD.

This class may require further subclassing if other methods are used in this framework for passing requests. However, this class has not been studied well enough to determine whether it uses additional classes for performing it's job.

2.3.9 "Controller" Class (Derived from BasicType)

This class defines the top level processing or "base" classes. RequestReceiver will find the Controller subclass type stored in the request in the database and in turn instantiate the Source, Sink, and RequestBuilder subclasses and do whatever is needed for initializing.

2.3.10 "Source" Class (Derived from BasicType)

This class is responsible for instantiating classes for reading data into the process (DMAD,DEAD, etc.) from file, RPC, TCP/IP socket, etc. and subclasses for accessing the data.

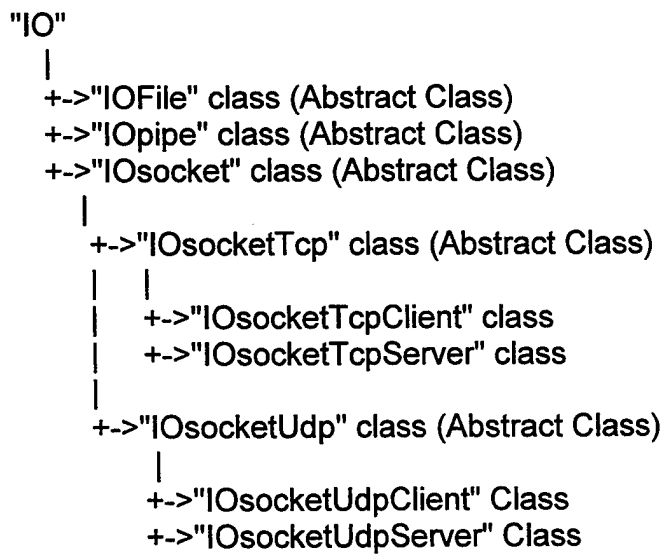
2.3.11 "Sink" Class (Derived from BasicType)

This class is a mirror image of "Source". It is responsible for instantiating classes that perform output.

2.3.12 "IO" class (Abstract Class and Derived from BaseType)

Subclasses of "IO" perform input and output of data into the process. One is instantiated at the Source (input) and one at the Sink (output). In a Broker architecture this approach may need to be slightly modified.

Some possible subclasses for "IO" may include:



2.3.13 "ChainProcess" Class (Abstract Class and Derived from BasicType)

This is the parent to a group of subclasses that process the data. The name implies that processing may be linked together using the attributes and information in the database.

The subclasses that follow are subclasses needed for Back End processing.

2.3.13.1 "Extraction" Class (Abstract Class and Derived from ChainProcess)

This is the parent to a group of subclasses that perform extraction of the data.

2.3.13.1.1 "ExtractMajorFrame" is a subclass of "Extraction" used to maintain a list of DataMinorFrames, calls another class to read in a specified amount of data, finds the next instance of DataMinorFrame which recognizes the data, and passes a copy of its object pointer out to output if the major frame was also requested (although unlikely).

2.3.13.1.2 "ExtractMinorFrame" is a subclass of "Extraction" used for "extracting" the minor frame pointed to by Major Frame and managing a list of channels and putting it's own data out to output, if this (extracted) minor frame was also requested.

2.3.13.1.3 "ExtractChannel" is a subclass of "Extraction" used for decommutating its channel pointed to by Minor Frame.

2.3.13.2 "Description" Class (Abstract Class and Derived from ChainProcess)

The subclasses of "Description" maintain information about how the data was originally designed to be processed. This normally applies to channel data but for completeness, Major Frame and Minor Frame are included here.

If allowed, it will instantiate those methods and classes for processing the data. Otherwise, the "Processing" subclasses may override the methods in the Description classes with its own methods specified within "Processing" class attributes.

2.3.13.2.1 "DescriptMajorFrame" is a subclass of "Description". It is included only for completeness.

2.3.13.2.2 "DescriptMinorFrame" is a subclass of "Description".

2.3.13.2.3 "DescriptChannel" is a subclass of "Description". DescriptChannel will describe (own or link to) one or more ExtractChannels. Owning or linking to two channels is one way to accommodate derived channels.

2.3.13.3 "Processing" Class (Abstract Class and Derived from ChainProcess)

This class exists solely for the purpose of providing a means of customizing display formatting performed on the DMAD.

2.3.13.3.1 "ProcessMajorFrame" is a subclass of "Processing". This class name is included only for completeness.

2.3.13.3.2 "ProcessMinorFrame" is a subclass of "Processing".

2.3.13.3.3 "ProcessChannel" is a subclass of "Processing". This class provides a means of customizing channel processing as in the case when an analyst customizes an instance of a display by changing alarm limits. This class owns or links to a "DescriptChannel".

2.3.13.4 "FormatData" Class (Abstract Class and Derived from ChainProcess". The subclasses of this class format data for including in the FormatDisplay class. The subclasses could be as follows:

2.3.13.4.1 "FormatMajorFrame" is a subclass of "FormatData". This class name is included only for completeness.

2.3.13.4.2 "FormatMinorFrame" is a subclass of "FormatData".

2.3.13.4.3 "FormatChannel" is a subclass of "FormatData". The subclasses of this class formats a channel for one of the FormatDisplay subclasses. This class links to or owns a "ProcessChannel". Some of the subclasses of this type may include the following:

"FormatChannelForDisplayList"
"FormatChannelForDisplayPlot"
"FormatChannelForDisplayModel"

2.3.13.5 "FormatDisplay" Class (Abstract Class and Derived from ChainProcess).

The subclasses of this class format a list of channels or other data type into frames and files according the attributes for a given instance of a display.

2.3.13.5.1 "FormatDisplayList" is a subclass of "FormatDisplay" for formatting frames of data into file for List displays.

2.3.13.5.2 "FormatDisplayPlot" is a subclass of "FormatDisplay" for formatting frames of data into file for plotting one or more channels.

2.3.13.5.3 "FormatDisplaySubsystemModel" is a subclass of "FormatDisplay" for formatting frames of data into file for displaying subsystem models.

2.3.14 "AlarmCheck"

This is an abstract class for performing Alarm Checking. Subclasses may include:

"AlarmCheckFltPnt"
"AlarmCheckNone"
"AlarmCheckStatus"

2.3.15 "ConversionChannel"

This is an abstract class for converting raw channel data into another data type. Subclasses may include the following:

"ConvertFltPnt"
"ConvertFltPntTable"
"ConvertFltPntcoef"
"ConvertFltPnt5coef" is a subclass of "ConversionFltPntcoef".
"ConvertFltPnt4coef" is a subclass of "ConversionFltPntcoef".

2.3.16 "TimeTag"

This is an abstract class for formatting and interpreting the time tag bits. This class was not analyzed but subclassing might be according to clock type.

"TimeTag_UTC" for time stamp in UTC
"TimeTag_Dsp" for DSP satellite clock time stamp
"TimeTag_Ufo" for UFO satellite clock time stamp

2.3.17 "DisplayFormatMethod" belongs to the strategy "offshoots" classes.

Subclasses may include:

"DisplayFormatMethod_Frame"
"DisplayFormatMethod_File ".

2.3.18 "RegisterForData" is a class for keeping track of classes in the DMAD "chain" during the build process so that an object can be used in more than one link at a time.

2.3.19 "ServerForData" is similar to "RegisterForData" except it applies to copying ExtractChannels just received from the DEAD to other ExtractChannels after first determining the 'instance of the subclass' of the first.

2.4 Software Tools

RogueWave provides tools and C++ classes for a low level framework which appear to be compatible with the framework developed in this report and tools . See web page http://www.roguewave.com/products/products.html for more information.

Tools.h++ Professional
DBTools.h++
zApp
ObjectFactory

Test

3.1 Description

The test was to implement a small demonstration to show proof of concept. The model chosen to demonstrate the validity of most of the approach stated in Section 1.3, was a simplified version of the DEAD. The code for this test was written in C++ and used the APIs to a RDBMS for accessing an account with schema and attribute data for extraction of channels and minor frames.

3.2 Model

The model used in the demo is illustrated in a figure titled "Proof-of-Concept Design". Notice that the names of classes and subclass hierarchy do not necessarily match the names given in the Framework or the DEAD model but the model that was used in the "Proof-of-Concept" is similar enough to the DEAD for verifying the approach described earlier. Print statements were used to show the stages of the build phase and the results of the decommutation process.

The database contained only a small subset of the rows needed for defining every channel in the DSP Link 2 frame.

Items in the approach used in the demo included:

(i) Use of integer types for assigning class types to fields in the database.(RTTI was not available in the compiler used to develop the demonstration so integers were hardcoded and a switch statement was used in the Factory for instantiating the correct class).
(ii) A modified Factory Design Pattern: "Factory"
(iii) Common Abstract Parent Class: "SomeAbsType" (named "BaseType" in the report).
(iv) Retrieval of attributes from the database by calling "DBMS" (named "DbmsIf" in the report) and passing an SQL string.
(v) A modified Chain of Responsibility Pattern through mechanisms mainly built into the common abstract class.
(vi) DBMS query for retrieving attributes: using SQL passed to "DBMS".
(vii) Use of the Strategy and Bridge Design Patterns in "DataMajorFrame" (or "ExtractMajorFrame" in report) to "DataMinorFrame" (or "ExtractMinorFrame" in report) and "DataMinorFrame" to "DataChannel" ("ExtractChannel" in report) through the use of the parent class "Data" ("Extraction" in report).

For the sake of meeting the schedule, the data in the database was limited to two types. The first was only defined for six channels in the DSP Link 2 which

resulted in entering 6x128 rows in the DataChannel (or ExtractChannel) table in the database. The second was a small made up example set of channels.

Note that one very slight design error was inadvertently introduced during coding and several shortcuts were taken but neither the shortcuts nor error detract from the objectives of this test.

See Figures 18,19, and 20 for models of the proof-of-concept demonstration.

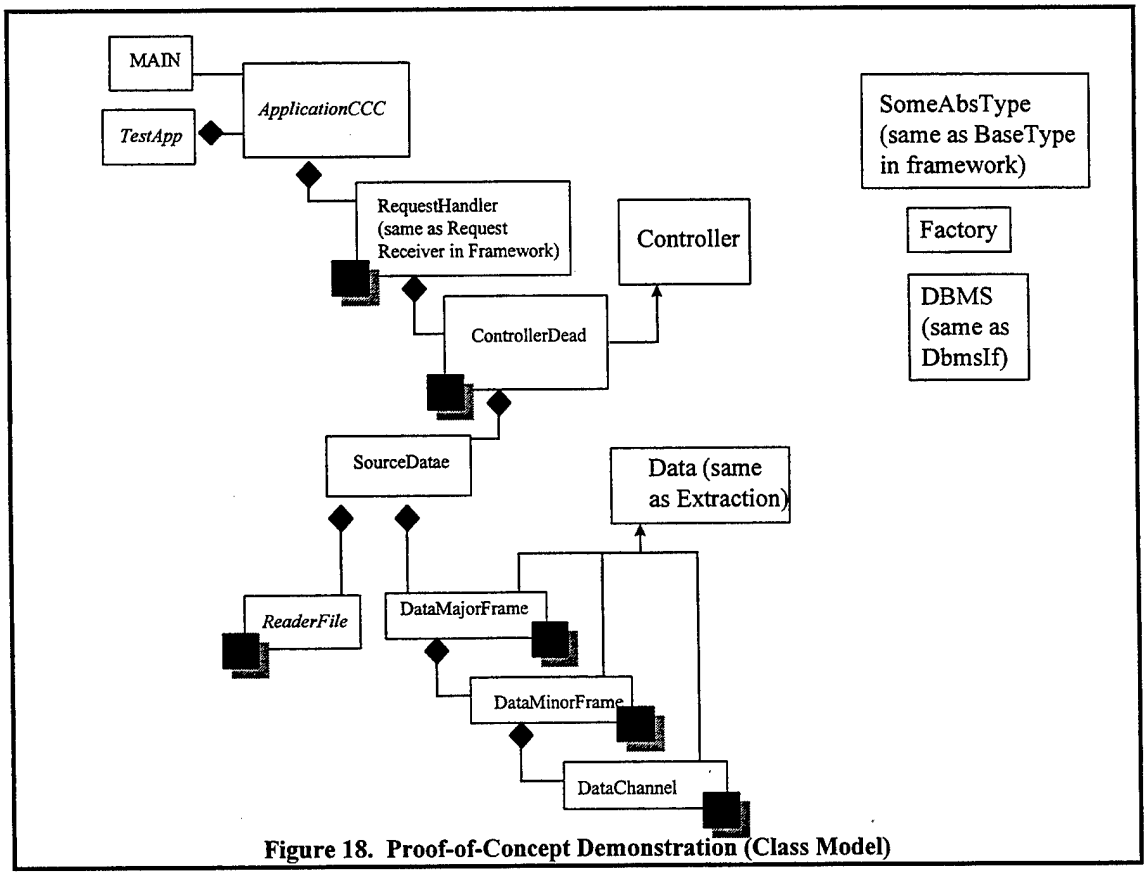


Figure 18. Proof-of-Concept Demonstration (Class Model)

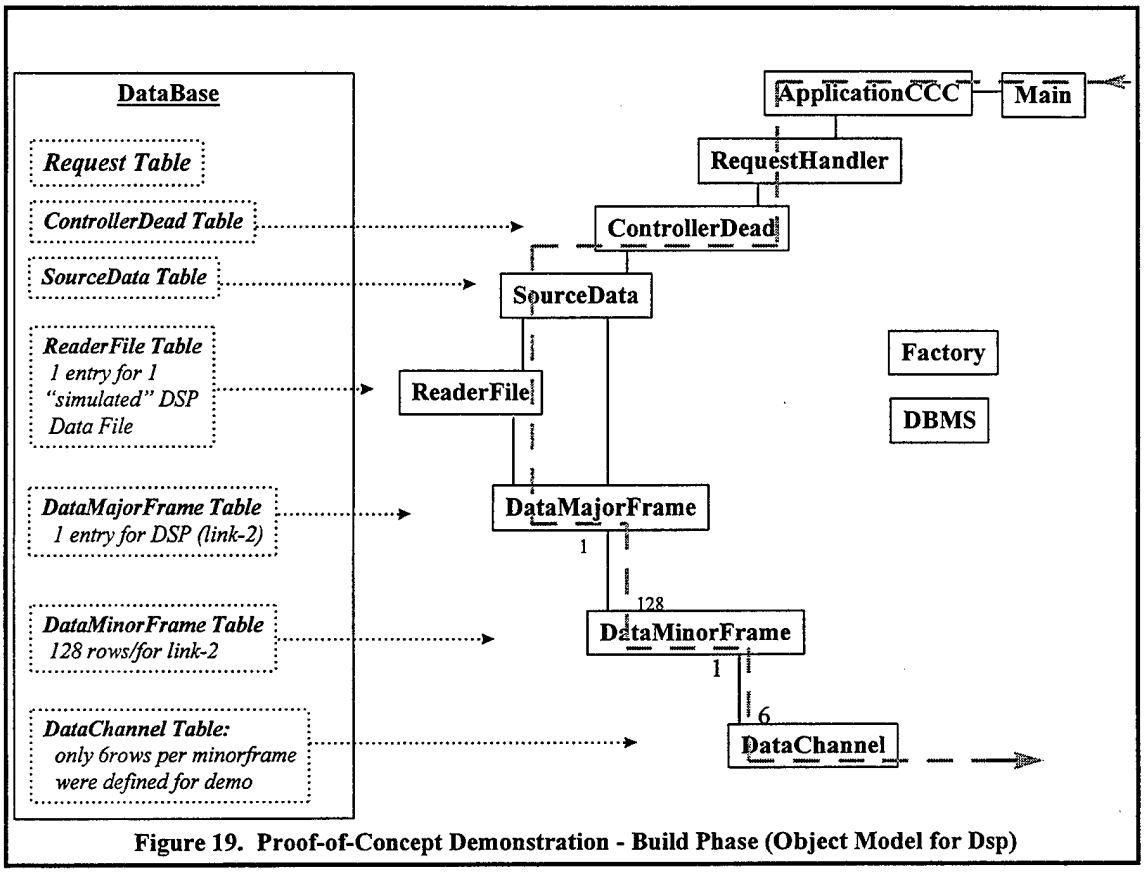


Figure 19. Proof-of-Concept Demonstration - Build Phase (Object Model for Dsp)

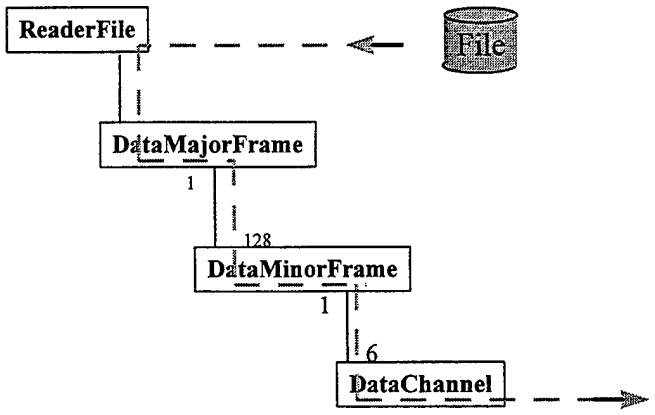


Figure 20. Proof-of-Concept Demonstration - Process Phase (Object Model for Dsp)

3.3 Results and Lessons Learned

All items included in the demonstrations, proved to work successfully. The build phase turned out to be much slower (several minutes) than expected while the process phase performed as well as expected. The amount of memory used appeared to be less than what was being used in some root processes running on the same workstation. However, if all the bytes in the DSP Major Frame were defined for channels, the DSP Major Frame would have produced 16,384 rows in the "DataChannel" Table (rows = 128frames x 128bytes), with an equal number of DataChannel objects instantiated. The build process would have taken longer to instantiate 20 times as many channels, and used almost 20 times as much memory. Assuming a database could support all the attribute data required by the model, the framework and model should be designed to support multiple accounts for storing satellite specific data in separate accounts and other data in a joint account. Obviously this issue requires more study.

Conclusions

The study described in this report shows that an extensible design for processing telemetry in a multi-satellite system can be developed using several commonly used object oriented design patterns combined with a new approach of using a DBMS for retrieving attributes for initializing an object during run-time. The original intent of using a database was to reduce the number of subclasses for some class hierarchies by generalizing classes and externalizing the attributes and to simplify the effort of extending a system by shifting the work from software maintenance to database management. After some analysis it became clear that this approach adds several subtle improvements over a non-database approach including reducing the complexity by using the attributes in one object to find and build the next object layer until the structure was complete. As long as no new classes or subclasses are needed, this instantiate-and-initialize one-class-at-a-time approach, built into the framework, allows the software to be extended simply by inserting the processing, stream, and display attributes into the database.

The framework, developed from these approach, is mainly for back end and front end processing and does not adequately address display or GUI designs. However, the approach could easily be applied to these areas using design patterns which are more appropriate to displays and GUIs.

Bibliography

Books:

Bamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John; "Design Patterns" Addison-Wesley, Reading MA. 1995

Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerlad, Peter; Stal, Michael; "A System of Patterns" Wiley New York, NY 1996

Eckel, Bruce "Thinking in C++" Prentice Hall, Inc., Englwood Cliffs, New Jersey 1995

Fowler, Martin and Scott, Kendall "UML Distilled" Addison-Wesley, Reading MA. 1997

Muller, Pierre-Alain "Instant UML" Wrox Press Ltd., Birmingham, 1997

Cooper, Richard; "Object Databases, An ODMG Approach" Thomson Computer Press, Boston MA, 1997

Glossary

1. Back End: As used in the report, this is the area of a telemetry system for extracting and processing telemetry data. It is performed after data has been processed by the Front End.

2. Channel. This is usually a few bits of information extracted from the stream for describing a sensor. This is also referred to as a mnemonic, sensor, word, data number (DN), etc.

3. Decommutate. This is the process of extracting the channel. from the stream.

4. Front End. This is the processing is usually associated with frame synchronization, decoding, decrypting, time tagging the data. This processing occurs before Back End processing.

5. Object Instance. This expression describes an object which has had it's attributes set for some specific subclass-like behavior.

6. Run Time Type Information. Also known as RTTI. This is a recent feature to some C++ compilers for converting class types and data types to integers and back again.

7. Sync Code. This is bit pattern at the beginning of a frame used for frame synchronization.

DISTRIBUTION LIST

AUL/LSE
Bldg 1405 - 600 Chennault Circle
Maxwell AFB, AL 36112-6424 1 cy

DTIC/OCP
8725 John J. Kingman Rd, Suite 0944
Ft Belvoir, VA 22060-6218 2 cys

AFSAA/SAI
1580 Air Force Pentagon
Washington, DC 20330-1580 1 cy

AFRL/PSTL
Kirtland AFB, NM 87117-5776 2 cys

AFRL/PSTP
Kirtland AFB, NM 87117-5776 1 cy

GenCorp Aerojet
P.O. Box 296
1100 West Hollyvale St
Azusa, CA, 91702-0296 1 cy

AFRL/VS/Dr Fender
Kirtland AFB, NM 87117-5776 1 cy

Official Record Copy
AFRL/VSSS/George Schneiderman
Kirtland AFB, NM 87117-5776 2 cys

SMC/CW
155 Discoverer Blvd
El Segundo, CA 90245-4692 1 cy

SMC/XR
180 Skynet Way 2234
El Segundo, CA 90245-4687 1 cy

Lockheed-Martin Astronautics
Flight Systems
Attn: Dr. Noel W. Hinners, MS S80000
12257 State Highway 121
Littleton, CO 80217 1 cy

TRW Space and Electronics
Attn: Joanne McGuire
1 Space Park
Bldg R10, Rm 2826
Redondo Beach, CA 90278 1 cy

Boeing
Attn: Mike Mott
2201 Seal Beach Blvd
Seal Beach CA 90740 1 cy

Hughes Space and Communications
Attn: Dr. Tom Brackey, MS S312
2260 E. Imperial Highway
El Segundo, CA 90245 1 cy