

# **IDC RE-ENGINEERING REPORT**

SAND20XX-XXXX

Unlimited Release

December 2015

## **IDC Re-Engineering Phase 2 Architecture Document**

**Version 0.1**

John Burns

Prepared by  
Sandia National Laboratories  
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.



**Sandia National Laboratories**

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

**NOTICE:** This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.



SAND201x-xxxx  
Unlimited Release  
December, 2015

# **IDC Architecture Document**

## ***Initial Draft***

J. Burns  
Next Generation Monitoring Systems  
Sandia National Laboratories  
P.O. Box 5800  
Albuquerque, New Mexico 87185-MS0401

### **Abstract**

This document contains a description of the system architecture for the IDC Re-Engineering Phase 2 project. This is a draft version that primarily provides background information for understanding delivered Use Case Realizations.

This page intentionally left blank

## TABLE OF CONTENTS

1	Introduction .....	7
2	Documents .....	8
2.1	Documents.....	8
3	Scope .....	9
3.1	System Overview.....	9
3.2	Mission.....	9
3.3	Deployment Concept .....	9
3.4	Re-engineering Principles .....	9
3.4.1	Re-Architect System Using Model-Based Engineering .....	10
3.4.2	Enhanced Mission Capabilities.....	10
3.4.3	Extensibility .....	10
3.4.4	Platform Independence .....	10
3.4.5	Integrated Testing.....	10
3.4.6	Incremental Transition .....	11
3.4.7	Modernize Development Process and Environment .....	11
3.5	Assumptions .....	11
4	Architectural Representation.....	12
4.1	Use Case View .....	13
4.2	Logical View .....	15
4.2.1	Control-Based Architecture .....	15
4.2.2	Domains.....	17
4.2.3	Analysis Classes.....	19
4.2.4	Types of Analysis Classes .....	19
4.2.5	Use Case Realizations .....	21
4.2.6	Mechanisms .....	21
4.2.7	Patterns .....	26
4.2.8	Key Features .....	37
4.3	Implementation View .....	40
4.4	Process View .....	40
4.5	Deployment View.....	40
5	Works Cited.....	41
6	Appendix A. Specifications .....	42

TABLE OF FIGURES

Figure 4-4. OSD Components .....24  
Figure 4-5. Relationship between Control Class and Plugin Class .....28  
Figure 4-6. Plugin Initialization.....29  
Figure 4-7. Plugin Invocation in Automatic Processing .....30  
Figure 4-8. Plugin Invocation in Interactive Processing .....31  
Figure 4-9. Event Analysis Classes.....36  
Figure 4-10. Event Hierarchy .....37

## 1 INTRODUCTION

The International Data Centre (IDC) of the Comprehensive Nuclear-Test-Ban Treaty Organization (CTBTO) applies, on a routine basis, automatic processing methods and interactive analysis to raw International Monitoring System (IMS) data in order to produce, archive, and distribute standard IDC products on behalf of all States Parties. The routine processing includes analysis of events with the objective of screening out events considered to be consistent with natural phenomena or non-nuclear, man-made phenomena. This document includes the dissemination of radionuclide data and products but not the acquisition or processing of radionuclide data.

The purpose of this Architecture Document is to describe the overall structure of the software and hardware included in the IDC Re-engineering project. The Architecture Document is the primary artifact created and maintained by the architecture team; it serves as the communication medium between the architecture team and other developers.

## 2 DOCUMENTS

### 2.1 Documents

<b>Author/Document Rev.</b>	<b>Title</b>	<b>Date</b>
IDC-RP2-SSD-V1.3	IDC System Specification Document	Dec 2015
IDC-RP2-UCMS-V1.1	IDC Use Case Model Survey	Dec 2015
IDC-RP2-GLOSSARY-V1.1	IDC Glossary	Dec 2015
—END OF TABLE—		

### **3 SCOPE**

The Architecture Document captures the general principles used throughout the design of the system and serves as a guide for the further development of the system. It is not meant to be a complete design document. To keep it short and accessible, and to avoid too much redundancy with other documents, this Architecture Document is intentionally limited to a high level of abstraction.

The Architecture Document evolves during the design of the system and continuously captures new major design decisions made regarding the system. In the context of an iterative development process, there is one version of the Architecture Document per development iteration.

The terminology and graphical notation used in this document are derived from the Unified Modeling Language (UML).

#### **3.1 System Overview**

The Comprehensive Test Ban Treaty Organization (CTBTO) operates the International Data Centre (IDC) to support the Comprehensive Test Ban Treaty (CTBT) monitoring mission. The IDC collects data produced by the network of sensors called the International Monitoring System (IMS). The IDC integrates, processes, and analyzes data in order to detect, locate, identify, and report natural and man-made events. It also collects and forwards, as required, data to National Data Centers (NDC).

#### **3.2 Mission**

The IDC's mission is to collect, analyze and distribute information to support monitoring compliance with nuclear test treaties by NDC's. This mission requires IDC to detect, locate, identify, evaluate, store, and report natural and man-made events. The IDC uses several different monitoring techniques to perform this mission, each designed to monitor a specific physical domain (e.g., space, atmosphere, underground, underwater) for events.

#### **3.3 Deployment Concept**

When the CTBT enters into force the System will support operations 24 hours per day, 7 days per week (24x7). The IDC is deployed at a single fixed site but the system supports multiple remote users as well as stand-alone, portable systems.

#### **3.4 Re-engineering Principles**

Re-engineering of the IDC is necessary to maintain current mission capability, reduce the cost of software maintenance, and provide the ability to enhance the system into the future. The following principles support achieving the goals of the modernization program.

- Re-architect System using Model-Based Engineering
- Enhanced Mission Capabilities
- Extensibility
- Platform Independence
- Integrated Testing

- Incremental Transition
- Modernize Development Process and Environment

#### ***3.4.1 Re-Architect System Using Model-Based Engineering***

Model-based engineering is a methodology that focuses on creating and maintaining a set of models that describe different aspects of the design of the system. During the design and development process the models provide an abstraction to facilitate discussion of significant system features. Models provide a means of capturing design decisions and rationale. Additionally models provide a basis for on-going support and enhancement of the system. Model-based engineering is a key to most modern software lifecycle processes.

#### ***3.4.2 Enhanced Mission Capabilities***

One of the primary objectives of this effort is to enhance existing data acquisition, event detection, and data distribution, and data retention capabilities to meet current and future treaty monitoring needs as specified in the IDC System Requirements Document (SRD). The design of the re-engineered system integrates improved seismic, hydroacoustic and infrasonic propagation models (velocity, attenuation, etc.) to improve association, location, magnitude estimation and event screening. Another area of focus is exploiting historic data captured by the system to improve automated system performance. The re-engineered system captures detailed histories of analyst interactions with the system to support refining the system and improving diagnostic capabilities.

#### ***3.4.3 Extensibility***

The System is designed to support continued expansion and refinement of the mission requirements. The system software is based on open standards and leverage software trends supporting extensibility. The system design facilitates integration of new tools and models to permit continued evolution and improvement of the system. Algorithms and models are key areas of improvement for the system so the design will describe interfaces and loose coupling of components to permit addition or substitution of these components. The system will also allow system maintainers to modify processing sequences and parameters to permit tuning system performance without extensive redesign and recoding. Another area of focus for the design is operator customization. The system will use COTS and user interface frameworks to permit users to tailor their displays to their work preferences so users can be efficient.

#### ***3.4.4 Platform Independence***

To facilitate continued system improvement and to avoid vendor-lock the System will emphasize open source operating systems and standard frameworks.

#### ***3.4.5 Integrated Testing***

A fundamental aspect of verifying the System will be the ability to reproduce a set of inputs to the system and capture results of processing that data. The system will capture raw input data and provide the capability to replay the data through the system. The system will capture provenance information about system performance, processing parameters and intermediate results to facilitate analysis and debugging.

#### **3.4.6 Incremental Transition**

The IDC shall work closely with development organizations in an integrated product team to re-engineer the System in a series of builds.

#### **3.4.7 Modernize Development Process and Environment**

A major objective of this effort is to make use of modern software design and development practices to re-architect the System software baseline that has evolved over several years. This effort shall include recoding the software baseline to take advantage of modern object-oriented software languages and 64-bit multi-core CPU architectures. The architecture design will utilize software modeling tools and processes to communicate, review, and maintain the architectural design of the system. Common tool sets at geographically distributed development site will facilitate a shared understanding and implementation of the architecture.

### **3.5 Assumptions**

The following assumptions were used in developing the architecture:

- The sensor network will be similar in type, size, and complexity as currently defined.
- The re-engineered IDC System will be operated as described in the IDC Operations Manual.
- The CTBTO will continue to operate a primary processing system at the Vienna International Centre .
- The Rational Unified Process will be used for the complete lifecycle of the project.
- The project will be a collaborative effort between the CTBTO and external development organizations. The CTBTO has allocated staff resources to support this effort.
- The CTBTO mission must execute fully and efficiently during transition to the new system.
- The new system must meet needs for the next ~20 years.

## 4 ARCHITECTURAL REPRESENTATION

Architecture is a concept that is easy to understand, but is hard to define precisely. In particular, it is difficult to draw a sharp line between design and architecture—architecture is one aspect of design that concentrates on some specific features.

The System is inherently complex. To reduce the complexity, we decompose the system into smaller components or *objects* through the use of Object-Oriented Design methodology. Objects abstract from the complexity by hiding the unimportant details and focusing on the important characteristics and operations.

The decomposition, abstraction and hierarchy can be developed along several orthogonal views, depending on the usage of the system.

- From the ground system application point of view, entities of the real world are mapped onto corresponding design entities; for example abstractions such as Sensors, Stations, etc.
- From the end-user's (operator's) point of view, certain inputs trigger actions to take place concurrently or sequentially throughout the system.
- From the system designer or programmer point of view, the system is organized in a way that helps its construction, that facilitates its development by several teams concurrently, that helps the long-term management of the software and hardware, and that facilitates the reuse of components throughout the system.
- From the operational and testing points of view, the system can be fully exercised according to the use cases and scenarios. A use case is a sequence of actions performed by the system that yields an observable result.

The system architecture attempts to capture and describe all these aspects rigorously and systematically. It offers multiple views or models of the software, each developed according to its own well-defined set of rules, each revealing one aspect of the system, but all consistent with one another.

In the Rational Unified Process (RUP) used to develop the System, analysis starts from a typical set of views, called the *4+1 View Model* [1]. It is composed of:

- **Use Case View** – Contains the use cases that encompass architecturally significant behavior, classes or technical risks. A use case defines the functions of the system by describing actor interaction with the system.
- **Logical View** – Realizes each use case through the use of high-level Analysis Classes depicted on Unified Modeling Language (UML) class and sequence diagrams. Analysis Classes describe a set of objects that share the same responsibilities, relationships, operations, attributes, and semantics.
- **Implementation View** – Organizes Analysis Classes into modules. The Implementation View addresses the organization of delivered source code modules in order to facilitate software development,

manage subsystem reuse and reduce subsystem dependencies. Modules are then mapped into layers; yet another level of organization intended to reduce the overall complexity of the software.

- **Process View** – Contains the description of the processes in the system, their interactions and configurations, and the allocation of Analysis Classes to processes. This view is needed because the system has a significant degree of concurrency.
- **Deployment View** – Contains the description of the various physical processing platform configurations, and the allocation of processes (from the Process View) to the physical platforms. This view is needed because the system is distributed.

In addition to the above views, the term *architecture* also includes the overarching patterns and/or frameworks that serve to shape the software.

#### 4.1 Use Case View

Use cases provide a basis for describing the system from a user-level or external perspective. Because use cases describe the functionality of the system from the user's point of view additional context or requirements are discovered from developing and reviewing the use cases. In turn, use cases provide a basis for organizing and analyzing the Logical View and Process View.

In a complex system such as this there are many use cases, some more complicated than others. An important job of the Architecture team is to determine up front which use cases are likely to have significant impacts on the final architecture of the system. Such use cases are referred to as *architecturally-significant* and should be architected into the system first to stabilize the architecture for later work. The determination as to whether a given use case is architecturally-significant or not is largely a subjective matter, but in general is based on answers to the following questions:

- Will the use case have a strong influence on how the overall system is architected (e.g., will it require a certain framework to be put in place, or certain widespread assumptions to be made)?
- Will implementation of the use case involve many architectural elements (many interfaces, processes, displays, etc.)?
- Does the use case represent an important (perhaps mission-critical) interaction with the system?
- Will implementation of the use case involve higher-than-average technical risk (e.g., excessive data rate or data storage, interfacing with a new external system for the first time)?

The following list shows the use cases defined as *architecturally-significant* for the System.

- **1.1 System Receives Station Data** – This use case is architecturally significant because it deals with the storage of data in the internal format and the timeliness for transferring data to the Data Processing Partition.

- **1.4 System Acquires Meteorological Data** - This use case is architecturally significant because it deals with acquisition, storage and processing of meteorological data.
- **2.3 System Detects Events using Waveform Correlation** – This use case is architecturally significant due to large amounts of waveform data being processed and the impact on waveform data storage and management.
- **2.6 System Builds Events using Signal Detections** – This use case is architecturally significant due to the potential of integrating new algorithms to build events.
- **2.12 System Predicts Signal Features** – This use case is architecturally significant due to the processing and memory resource consumption of 3D earth model calculations.
- **3.2 Refines Event** – This use case is architecturally significant because it captures the interplay between all of the Analyst activities.
- **3.2.8 Compares Events** – This use case is architecturally significant due to the introduction of the capability to compare events within an operational context.
- **3.3 Scans Waveforms and Unassociated Detections** – This use case is architecturally significant due to the user interface design considerations for displaying and interacting with waveforms and unassociated signal detections.
- **3.5 Marks Processing Stage Complete** – This use case is architecturally significant due to expected changes to the analysis process (including operational concept changes).
- **5.3 Views System Results** – This use case is architecturally significant because it provides an interactive interface for external customers.
- **6.3 Defines Processing Sequence** – This use case is architecturally significant due to the fundamental changes to the system it introduces including the interactive interfaces provided and the specification of the elements that are available for sequence customization.
- **6.7 Views System Configuration History** – This UC is architecturally significant because it covers a new feature of the system to store and view all system configuration information.
- **7.1 Analyzes Mission Performance** – This use case is architecturally significant because it provides the interface for measuring System event detection performance and tuning the system.
- **8.2 Controls the System** – This use case is architecturally significant due to the System's timeliness requirements to start and stop the System and to transfer mission assignment from the Primary to the Backup.

- **8.5 Views Event History** – This UC is architecturally significant because it covers review of stored versions of event hypotheses.
- **9.4 Replays Test Data Set** – This use case is architecturally significant due to the introduction of new capability and the concept of playing back time synchronized captured data.
- **11.2 Develops New Algorithms and Models** – This use case is architecturally significant as the Researcher requires access to System data and algorithm implementations used in pipeline processing and interactive processing through command line interfaces and a Common Object Interface (COI).
- **13.2 Performs Standalone Analysis** – This use case is architecturally significant since it requires the System architecture to support configurable software distributions at various scales of data processing, computing hardware, and personnel.
- **14.5 Performs Expert Technical Analysis** – This use case is architecturally significant due to the inclusion of data and software provided by member states.

## 4.2 Logical View

The Logical View depicts Analysis Classes and their behavior in the context of specific use cases. The functionality described in a use case is mapped to Analysis Classes in Use Case Realizations. The following sections describe Analysis Classes identified in the architecture and the frameworks in which they operate.

### 4.2.1 Control-Based Architecture

A concern in this effort is the requirement to decouple applications so they can be developed and replaced without affecting other parts of the system. Processes within the System may be initiated in two ways: automatically, in response to new or changed data, or interactively, in response to user commands. The system initiates automatic processing to analyze station data, detect signals and group signal detections into events. System users invoke interactive processing to also detect signals and group signals into events. Further analysis is performed to estimate the location and magnitude of events and screen based on the event source type. After interactive processing is completed, the system may initiate further automatic processing to compute other supporting information for the events formed by the operators. Similar analysis is performed in automatic and interactive processing so the system is designed to invoke the same processing components during automatic and interactive processing.

Another design concern is that the system should facilitate the modification or replacement of processing components and permit the processing components to be reordered or linked in alternate sequences. These capabilities are supported by decoupling processing components to the maximum extent and limiting the communication between components. The Processing Sequence Control Mechanism initiates automatic processing in **Control** classes based on rules defined by privileged users. **Control** classes operate on data stored in the

system and store the processing results back in the system, limiting the information passing between **Control** classes.

The approach taken in here is to employ a concept called Control-Based Architecture (CBA). In essence, CBA dictates that **Control** classes are the logical units of control in the system. In more formal terms, CBA is embodied by the following principles:

- All significant application logic is encapsulated by **Control** classes (not **Utility**, **Entity** or **Display** classes).
- **Control** classes are started and stopped by an external controller program, either the System Control Mechanism or the Processing Sequence Control Mechanism depending on the lifecycle of the **Control** class.
- **Control** classes retrieve inputs from the OSD, delegate algorithmic computations to **Plugin** classes, and store results in the OSD.
- **Display** classes retrieve data from the OSD.
- **Control**, **Display**, and **Plugin** classes are the only classes that utilize IPC.

Adherence to these principles provides several important benefits, including the following:

- Provides encapsulation of application logic.

**Control** classes are designated as the controllers for well-defined portions of application logic (e.g. event location, event magnitude). The Processing Sequence Control Mechanism initiates the **Control** classes and coordinates application logic among the **Control** classes. This approach limits the dependencies between **Control** classes. **Control** classes retrieve data from the OSD, implement processing logic either directly or via delegation to **Plugin** or **Utility** classes, and then return results to the OSD to signal the Processing Sequence Control Mechanism that the processing unit is complete. Encapsulating all of the sequencing responsibility in the Processing Sequence Control Mechanism rather than spreading that responsibility across all the **Control** classes supports the ability to fully configure processing sequences.

- Application logic can be relocated easily.

In essence, each **Control** class represents a relocatable unit of application logic which can be located in any process, as needed. A given process might contain multiple **Control** classes if each has little processing to perform. Alternatively, a process might be dedicated to a single **Control** class to ensure maximum single-processor performance. The ability to relocate application logic as needed is a key capability for providing the flexibility to meet unknown future processing requirements with minimal source code changes.

- Class dependencies are greatly simplified.

**Control**, **Display** and **Plugin** classes perform communication, if required; though Interface classes. Therefore **Control**, **Display** and **Plugin** classes never depend directly on one another. Besides facilitating relocatability (as described in the previous bullet), this results in a reduction in dependencies between classes. This reduction in dependencies, in turn, facilitates the ability to reorder or replace processing units.

#### 4.2.2 Domains

The System is composed of the following domains. These domains provide a logical organization for the Analysis Classes of the system.

- **Data Acquisition** – The Data Acquisition Domain contains classes for handling data reception from various data providers, forwarding the data, and storing the data for subsequent access by the IDC System. The System acquires data from stations, from external data centers, and from other sources. Data can be acquired in a variety of formats including CD-1.0, CD-1.1, IMS-1.0, SEED, and miniSEED.
- **Data Quality** – The Data Quality Domain contains classes for detecting errors in incoming waveforms that can lead to problems with processing and analysis. The data containing errors is stored and masked to prevent further processing. System users can override the system’s quality determination. Data quality errors include data gaps, amplitude spikes, repeated amplitude values, linear trends and invalid gain.
- **Signal Enhancement** – The Signal Enhancement Domain contains classes for applying signal processing techniques to enhance the signal content and reduce the noise content of waveform data. The techniques include filtering, beamforming, and three component waveform data rotation.
- **Signal Detection** – The Signal Detection Domain contains classes for using various techniques to identify signals of interest. Signal detections are stored and further processed to identify events.
- **Signal Feature Measurement** – The Feature Measurement Domain contains classes for measuring features associated with a signal detection, e.g. arrival time, back azimuth, horizontal slowness, amplitude, frequency content, etc. The feature measurements are used to analyze the signal detection.
- **Signal Detection Association** – The Signal Detection Association Domain contains classes for using observed and predicted signal detection features to associate signal detections with either new events or existing events.
- **Waveform Correlation** – The Waveform Correlation Domain contains classes for finding new events by matching current waveforms to waveforms of known historical events. A matching new event is created at the location of the historical event.
- **Event Conflict Resolution** – The Event Conflict Resolution Domain contains classes for resolving cases where signal detections are assigned to more than one event. Each signal detection should be associated to at most one event.

- **Event Location** – The Event Location Domain contains classes for determining an event's spatial location and temporal location and uncertainties.
- **Event Magnitude** – The Event Magnitude Domain contains classes for estimating the size of an event by combining the available event magnitudes computed from individual stations.
- **Moment Tensor** – The Moment Tensor Domain contains classes for applying long period waveform modeling to determine a moment tensor representation of the source of seismic events. The moment tensor quantifies both the event size, and the event type (earthquake vs. explosion).
- **Event** – The Event Domain contains classes for tracking general event information not addressed by other domains. This includes classes for tracking event version history information, finding and tracking events of interest, performing event searches, and computing event quality.
- **Performance Monitor** – The Performance Monitor Domain contains classes for tracking both system performance and monitoring mission performance. The system performance is characterized by disk usage, CPU load, network traffic, etc. The mission performance is characterized by waveform data availability, station signal detection rates, network event detection rates, etc.
- **Signal Feature Prediction** (includes physics models and historical models) – The Signal Feature Prediction Domain contains classes for calculating predicted values and uncertainties for observables associated with particular source to receiver paths. E.g. seismic travel time, azimuth, and slowness.
- **Geospatial Processing** – The Geospatial Processing Domain contains classes that access information that identifies the geographic location and characteristics of natural or constructed features and boundaries on the Earth, typically represented by points, lines, polygons and/or complex geographic features, and may contain information attached to a location. Geospatial data is often accessed, manipulated or analyzed through Geographic Information Systems.
- **Station** – The Station Domain contains classes that define the installation where monitoring sensors are installed. Multiple sensors can be installed at the same station. An array is a group of stations.
- **System Configuration** – The System Configuration Domain contains classes that describe the complete set of system parameters that define the operation of the System software. Examples include sensor thresholds, filters (see filter, waveform), the particular version of an earth model in use and processing sequences. Each instance of a system configuration is saved so the state of all parameters at any time can be recalled.

- **Process Control** – The Process Control Domain contains classes that define the configuration and sequencing of processing components in the system.
- **Data Distribution** – The Data Distribution Domain contains classes that provide access and distribute data to external customers of the system.
- **Testing** – The Testing Domain contains classes that create, store, and run system tests, compare system test results to expected results, and test the system via replay.

#### 4.2.3 Analysis Classes

Analysis Classes are the fundamental building blocks of use case realizations. The Use Case Realizations map the system functionality described in Use Cases onto the Analysis Classes. Through this process each Analysis Class contains a high-level description of the functionality the class is responsible to implement. The Analysis Classes in turn become the foundation for organizing the software implementation.

#### 4.2.4 Types of Analysis Classes

The class stereotypes below are used to designate the responsibilities of the analysis classes used in modeling Use Case Realizations. The stereotypes used are **Control**, **Plugin**, **Utility**, **Interface**, **Plugin Interface**, **Boundary**, **Entity**, **Display**, **Configuration** and **Mechanism**. Numerous other stereotypes are possible at the implementation or source code level (e.g., *proxy*, *adaptor*, *singleton*, etc.). At the architectural level, however, the concern is only with describing the essential building blocks of the system at a rather broad, high level. Thus, within the architecture, nearly all Analysis Classes fall into one of the categories below. In the architecture, all classes are categorized into one of the following stereotypes:

- **Control** – A class that operates in an event-driven manner and encapsulates some significant piece of logic, typically application-level logic. **Control** classes may be instantiated as a separate process or set of replicated processes (e.g. for performance). **Control** classes are designed to support separate instantiation but may in fact be combined into processes with other classes (e.g. **Display** classes, other **Control** classes) for performance reasons. The specific mapping of control classes to processes is specified in the Process View.
- **Plugin** – A class that encapsulates an isolated portion of the system which can be updated independently. Plugins may identify portions of the system for which multiple different implementations exist, such as key algorithms. A **Plugin Interface** class defines the common interface for all implementations of **Plugin** class behavior. **Plugin** classes are designed to have simple interfaces to facilitate development and integration of new implementations while limiting the impact to the remainder of the system. To support this relative isolation, a **Plugin** class may only depend on **Plugin Interface** classes and the OSD. **Plugin** classes are highly scalable and configurable to meet system performance constraints. **Plugin** classes may be deployed in the same

process as the **Control** class using the **Plugin** or in a separate process. The specific mapping of **Plugin** classes to processes is specified in the Process View.

- **Utility** – A class that encapsulates program logic, sometimes at the application-level. Unlike **Control** classes, **Utility** classes are not designed to be instantiated as separate processes; instead, they are designed to be collocated in the same process with **Control** or **Display** classes. Their purpose is to assist that **Control** class in carrying out its function. A given **Utility** class may be used by several **Control** classes. Examples are math libraries and System Clock.
- **Interface** – A class that abstractly represents a defined interface to a class. Unlike **Control** and **Utility** classes, **Interface** classes have no intrinsic behavior built into them; they are simply used to describe messages and data communicated between classes that may potentially be instantiated in different processes. **Control** classes send data to other classes that may exist in separate processes via interfaces by *using* and *realizing* interfaces. In particular, a **Control** class that sends messages according to an interface is said to *use* that **Interface**, while a class that receives messages according to an interface is said to *realize* that **Interface**.
- **Plugin Interface** – A class that abstractly represents the defined interface to a **Plugin** class. A **Plugin Interface** class is similar to an **Interface** class but the **Plugin Interface** class is only realized by a **Plugin** class. Classes invoking a plugin always communicate with the **Plugin** class through a **Plugin Interface** class rather than directly calling operations on the **Plugin** class.
- **Boundary** – A class that abstractly represents an external system or actor (i.e., user). Much like **Interface** classes, **Boundary** classes are abstract and therefore possess no intrinsic behavior. **Boundary** classes can thus be viewed as a special kind of **Interface** class; one in which the sender or receiver is always external to the System. **Boundary** classes may represent any external actor including users, device interfaces, or machine-to-machine interfaces (TCP/IP, File Transfer Protocol (FTP), etc.).
- **Entity** – A class that encapsulates data rather than logic. **Entity** classes are typically simple classes for holding data. Because their internal state is fully self-contained, they may be persisted in a database or passed between processes as arguments of inter-process function calls.
- **Display** – A class that represents a user interface display. **Display** classes are similar to **Control** classes in that they run in an event-driven manner, and may be instantiated within processes in various combinations (as specified in the Process View). However, unlike **Control** classes, **Display** classes are dynamically created as needed according to events (e.g., a button press), a distinction important

enough to warrant the separate category. **Display** classes also can be started and stopped by an external control process.

- **Configuration** - A class representing a set of related configuration settings defining the default algorithm parameters used when Control and Plugin classes are invoked. Configuration classes support the ability to define parameters based on geographic region, time of year, time of day, network, station, channel, phase, observable type and processing stage. Configuration classes contain version information such as installation time, the system release that included the configuration change, etc. Configuration classes may be grouped into logical collections (e.g. processing sequence configuration, station processing configuration, location configuration, etc.) to organize the settings into general categories. This makes it easier for System Users to navigate the System configuration to find particular configurations. The System Maintainer sets configuration settings offline and installs them on the system.
- **Mechanism** – A Mechanism class representing a basic service or framework required by many subsystems across the system. Examples include Inter-process Communications, Processing Sequence Control, or Object Storage and Distribution—fundamental components which make up the framework upon which the application is constructed.

#### 4.2.5 Use Case Realizations

The process of analyzing use cases (which have an outward focus) and elaborating how the system will accomplish them internally via cooperating Analysis Classes is a process referred to as *use case realization* and is the primary activity performed by the Architecture team. The result of this activity is a complete and detailed set of use case realizations covering the entire set of use cases defined for the system.

Use Case Realizations are particularly important in the development of architecture. A Use Case Realization describes the collaboration of analysis classes to implement the functions defined in the Use Cases. The collection of Use Case Realizations in turn define the structure of the architecture needed to implement the significant features of the system. Use Case Realizations also help identify common patterns of interaction which form the basis for defining architectural patterns and mechanisms. Once the Use Case Realizations are developed they become the basis for detailed software design.

#### 4.2.6 Mechanisms

##### 4.2.6.1 System Control Mechanism

In a distributed system it is important to ensure all processes are fully initialized and ready to function before allowing data to flow into the system. Processes may need to interact with other processes to complete initialization. Similarly, when powering down it is important that processes are staged-down in a controlled manner to ensure the system is not left in an inconsistent state. Because of these issues, a mechanism for coordinating the startup and shutdown of processes across the system is required. This mechanism is known as System Control. This mechanism also monitors processes to ensure they are executing properly.

#### 4.2.6.2 Processing Sequence Control Mechanism

The System provides the capability to execute pre-defined processing sequences for automatic data processing. This capability is supported by the Processing Sequence Control mechanism, which is responsible for managing the execution of processing sequences based on definitions installed in the system. The Processing Sequence Control mechanism and select Processing Sequences are described further in the IDC Use Case Realization reports.

The Processing Sequence Control mechanism executes Processing Sequences based on triggering events in the system. Example triggers include the following:

- **Timer events** – Processing Sequences may be executed at pre-configured times or intervals (e.g. periodically checking for new waveform data to process).
- **Service Invocation** – Processing Sequences may be executed based on invocation of the Processing Sequence Control mechanism's service interfaces (via API or message-based service call). This type of trigger supports execution based on operator commands and other events in the system – e.g. for post-processing of created/modified data entities (signal detections, event hypotheses, events, etc.), processing stages, etc.
- **Data Subscription Callbacks** – The Processing Sequence Control mechanism maintains subscriptions for select data updates in the system that require a processing response (e.g. the creation of a new event). These subscriptions and the corresponding Processing Sequence(s) are installed as configuration items in the system. When the Processing Sequence Control mechanism receives callbacks for configured data subscriptions, it invokes the associated Processing Sequence(s).

The Processing Sequence Control mechanism supports a scalable, distributed processing model for execution of processing sequences. As depicted in **Error! Reference source not found.**, Tasks executed by the Processing Sequence Control mechanism may be implemented as service invocations routed to control classes running in separate processes, potentially on separate hosts within the system. This approach allows for parallel execution of Activities within a Processing Sequence across multiple processes and nodes.

#### 4.2.6.3 Object Storage and Distribution Mechanism

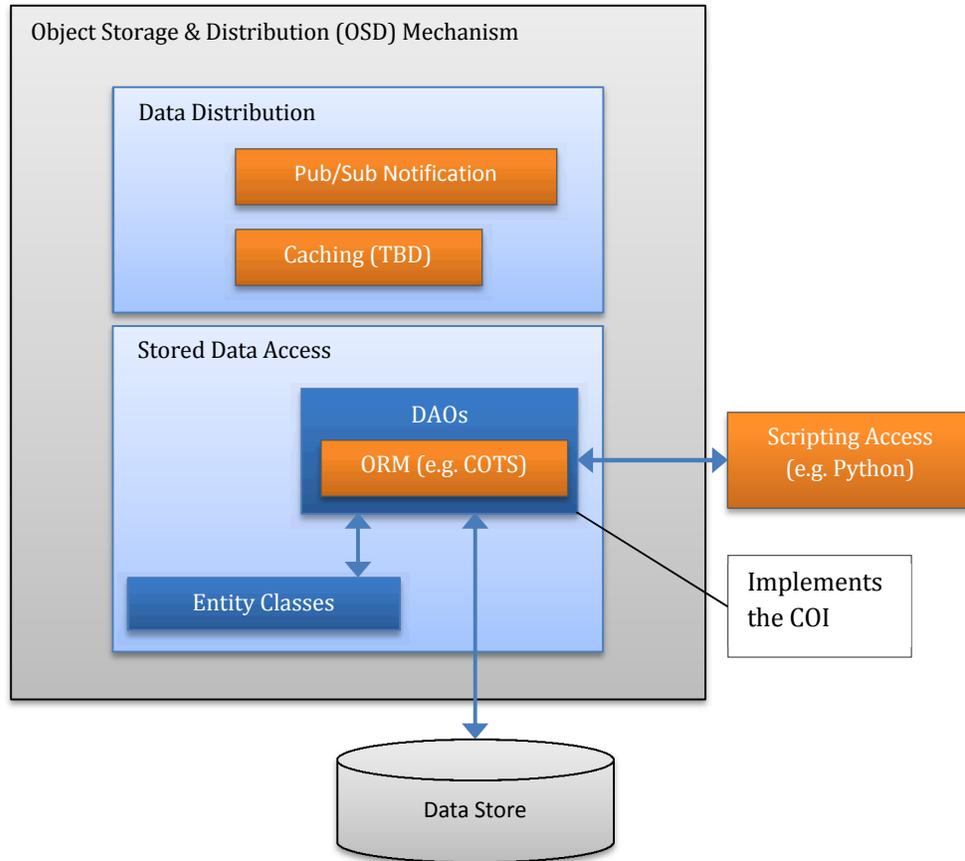
A central component of the System is the persistent storage of an extensive history of sensor and station data, station configuration data, significant signal detections and events. This information is stored in a RDBMS and, in previous versions of the system; access to the data was achieved by directly interacting with the RDBMS query language. This direct form of access resulted in strong dependencies between the application software and tools and the database structure, resulting in significant impact when modifying or expanding the underlying database schema. A design goal for the re-engineered system is to isolate the

application software from the underlying database schema to limit the impact of changing the database structure.

This isolation will be achieved through a mechanism called the Object Storage and Distribution Mechanism or OSD. The OSD will be responsible for persisting and retrieving data in the System. SQL calls from software applications will be replaced by calls to the OSD to retrieve the data. The OSD will be responsible for translating the data calls into queries on the underlying RDBMS. This pattern of abstracting the database interface is known as Data Access Object design pattern. The definition of the interface to the persisted data is described in the Common Object Interface (COI).

The OSD will also provide a subscription service to notify any software application that has registered interest in a data object when the data object is modified. Data often needs to be pushed or *distributed* to interested clients (*subscribers*) at the time that it is stored in the database (i.e., *persisted*), a pattern commonly referred to as *publish/subscribe*. The OSD may cache data in memory in addition to database storage to decrease access time to retrieve the data. The OSD will support the typical database functions Search, Create, Retrieve, Update, Delete (or SCRUD). The Delete operation will be limited to only system administrators. Instead of delete other operators will be able to mark an object as removed. Removing an object will mark the object as invalid but keep a copy in storage for review or further analysis.

The following figure illustrates the components of the OSD Mechanism developed during prototyping in Iteration E2. The mechanism provides to basic functions: Stored Data Access and Data Distribution. The figure shows the interaction of Data Access Objects with **Entity** Classes and the ORM to provide data storage and access. The figure also shows access to the stored data via scripting languages. In addition the figure shows Data Distribution implemented via Publish/Subscribe Notifications or via Caching. This model of the OSD Mechanism is the basis for the executable architecture and will evolve from further prototyping.



**Figure 4-1. OSD Components**

#### 4.2.6.4 Inter-Process Communication

As described previously, the OSD Mechanism is the primary means for exchanging data in the System. When communication is required between **Control**, **Display** and **Plugin** classes the communication will be implemented via inter-process communication or IPC. Because of the highly distributed nature of the System, limiting the use of IPC will support modifying the processing sequence and replacing **Control** or **Plugin** classes. When used, the IPC mechanism supports three flavors or patterns of communication, described below:

- Asynchronous** – A type of call where the caller does not block when a message is sent. This allows the caller to continue processing incoming messages immediately, however, the caller does not have any knowledge as to when or even if the receiving process actually received the message. In fact there could be any number of receiving processes listening for the message, including none. Because communication is essentially one-way, there is no way for the receiver to provide a return value for the message (other than sending a separate return message, independent of the first). This type of communication is sometimes described as fire-and-forget or broadcast. A limitation of this type of communication is messages could be lost if the receiver process crashes and if the sending process does not buffer the message.

- **Synchronous** – A type of call where the caller is blocked when a message is sent until the receiving process receives the message and sends back a return value. In this type of call there must be exactly one and only one recipient. If the recipient is not there or fails during processing of the message then the caller, is blocked waiting for the call to finish. For this reason Control classes tend to avoid this type of communication unless required.
- **Request/Reply** – A type of call where the caller is not blocked, but can receive return data from the recipient at a later time via a callback. This pattern of communication is used frequently because it provides the benefits of both **Asynchronous** messaging (no blocking) and **Synchronous** messaging (return of a value). In particular, **Request/Reply** communication is used heavily for communication between **Display** classes and **Control** classes. In this situation the **Display** class typically makes a request and then displays the result when the reply is subsequently received. This pattern is ideal for the user interface, which cannot afford to be blocked waiting for the reply but also needs to provide confirmation or feedback to the user regarding the request.
- **Publish/Subscribe** – In a publish/subscribe application, senders publish messages to a named topic that serves as a routing key for messages. Consumers may subscribe to one or more publish/subscribe topics in order to receive messages published to those topics. All subscribers to a given topic will receive copies of every message sent to that topic. Publish/subscribe is typically implemented as an Asynchronous pattern where publishing and receiving messages are decoupled.

Two messaging technologies have been selected to support IPC for the Executable Architecture Prototype.

- **AMQP Messaging** – The Advanced Message Queuing Protocol is a widely-used wire protocol standard for message oriented middleware. It supports all of the messaging patterns described above. RabbitMQ is a highly popular, AMQP-compliant, open-source messaging solution that has been selected for use in the executable architecture prototype. Specifically, RabbitMQ will be used to support publish/subscribe distribution of data in support of OSD subscriptions, as well as for network IPC service calls among control classes, displays and mechanisms. The selection of RabbitMQ is based on favorable benchmark performance, cross-language support, and its prevalence among messaging solutions.
- **REST-ful Webservices** – As a second network IPC option, the executable architecture prototype will include a REST/HTTP solution. This technology may be used as an alternative for remote service invocation using the claimcheck pattern for data passing (see Section 4.2.7.1 for more on the claimcheck pattern). In addition, the prototyping effort will explore the use of this technology for external access to data stored via the OSD, providing a language-agnostic data-as-a-service interface. The Jersey open-source framework has been selected to support this type of communication for Java software. Frameworks for other languages have not yet been selected.

#### 4.2.7 Patterns

Patterns describe common architectural approaches to addressing various tasks in the system. Patterns describe common interactions between the application software and the basic mechanisms. These patterns emerge from development of the Use Case Realizations and provide some of the fundamental building blocks for the system architecture.

##### 4.2.7.1 Data Access

**Control** classes interact with the Object Storage and Distribution Mechanism to access persistent data in the system. The OSD Mechanism implements *get* and *store* operations that provide access to data. The **Control** class passes the attributes identifying the object to retrieve to the OSD Mechanism and the requested object is returned to the **Control** class. The **Control** class modifies the requested object and then calls the OSD Mechanism to store the object. The **Control** class is not dependent on the underlying persistence method or database. The OSD Mechanism also provides a local store operation that updates the object but the update is not visible globally. The selection for storing globally or locally is defined in a Processing Context object that is passed to the OSD Mechanism as part of the store operation. The OSD Mechanism can also notify **Control** or **Display** classes when an individual object or collection of objects is modified. **Control** or **Display** classes register a callback for the object or collection of interest with the OSD Mechanism and the OSD calls the callback when the object or collection is modified. A common example occurs when a **Display** class provides a list of events in the system. The **Display** class registers a callback with the OSD Mechanism for all events. The OSD Mechanism calls the **Display** when new events are created or modified and the **Display** class updates the user interface. **Control** or **Display** classes unregister when the notification is no longer required.

For subscription-based OSD data callbacks that occur over a network (e.g. between processes &/or nodes), the executable architecture prototype will use a pattern similar to the *Claimcheck* Enterprise Integration Pattern (EIP). [2] Rather than serializing and transmitting the data entity to subscribing processes directly, under this pattern, the OSD will store the entity, making it globally accessible, and will transmit a 'claimcheck' message to subscribers. The claimcheck message will include reference information sufficient for the subscriber to retrieve the entity from the OSD upon receipt of the message. This pattern provides an efficient means of communicating data between processes, even for large data entities.

##### 4.2.7.2 Display Generation

**Display** classes are the primary point of interaction between the user and the System. They are how the System presents information to the user, and how the user provides input to and requests actions on the System. **Display** classes are typically opened when requested by the user and closed when the user has completed the interaction. **Display** classes request and receive data from the OSD either by directly requesting information or subscribing for updates. **Display** classes synchronize their views with changes made to the underlying data model. **Display** classes also pass user requests to **Control** classes through IPC.

The design of the Analyst workspace depends on Use Case Realization **Display** classes, Use Case Storyboard mockups, and Prototyping class implementations for Analyzes Event UC, Refines Event UC, Scans Waveforms and Unassociated Detections UC, and several other children UCs. Each **Display** class is responsible for different subsets of information that

make up the workspace. For example, the Analyzes Event Display is responsible for displaying the list of events, the Selects Data for Analysis Display displays the operator workflow, the Refines Event Display shows the detailed information about an individual event, the Enhances Signals Display shows the waveforms during the time interval of an event, and the Detects Signals Display shows the signal detections associated with an event.

In Use Case Realizations, **Display** classes are primarily modeled one per use case to capture a summary of the information transferred via the user interface for that use case. **Display** classes are also modeled to reflect common user interface functionality shared between Use Case Realizations. **Display** classes do not represent the actual layout of screens or windows in the user interface, and are presented as a means of communicating the interactions between users, **Display** classes, **Control** classes, and **Mechanisms**(like the OSD).

The design and layout of **Display** classes along with input action behavior are addressed in Use Case Storyboards. Storyboard mockups aim to provide a visual representation of the flow of action between a user and the System for a given Use Case.

Actual class implementations of **Display** classes are handled in Prototyping. Here, functionality, visual appearance and layout described by Use Case Realizations and Use Case Storyboards materialize as constructed class objects created in the chosen development environment that best suits the needs and requirements of the System. Prototyping also takes this development time to continue exploring risks, relationships, and details not addressed in Use Case Realizations and Use Case Storyboards. Prototyping is currently realizing **Display** class design and behavior using the Netbeans Rich Client Platform (RCP). The Netbeans RCP Prototype leverages capabilities provided by RCP and Java for customizable **Display** class layout, modular plugin framework, context-sensitive actions, and **Display** class data synchronization (See Draft Netbeans UI Design Notes for more details).

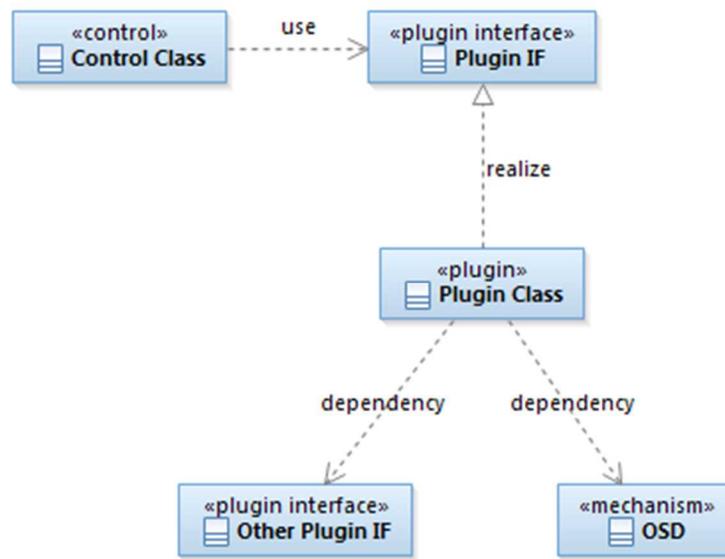
Further prototyping has been done to explore the possibility of web-based **Display** classes, accessible via a standard browser or browser-based tool. The Ozone Widget Framework (OWF) is a web framework for managing UI layout. It allows for multiple web views to be laid out in a customizable fashion. **Display** classes then consist of an HTML/JavaScript view loaded inside of OWF. Access to **Control** classes would occur via RESTful AJAX-based web-services against a server back-end. Data synchronization between views can occur through WebSocket technology, a web standard allowing for full-duplex communication between the server and browser client. Using WebSockets, changes to data can be pushed onto the browser views, which can update accordingly.

#### 4.2.7.3 Algorithms for Automatic and Interactive Processing

The System has the requirement to use common algorithms during automatic and interactive processing. This is achieved by the Processing Sequence Control Mechanism invoking the **Control** class for automatic processing and a **Display** class invoking the same **Control** class for interactive processing. The System Maintainer sets the default **Configuration** used during automatic processing while Analysts can select to override the settings used during interactive processing. When first started by System Control both **Control** and **Plugin** classes use **Configuration** classes to determine their default settings. The **Control** and **Plugin** classes will use these defaults for any settings that are not explicitly overridden in a particular call to the **Control** or **Plugin**. The system associates processing results (e.g. events, signal detections) with the settings used to create those results. Where applicable, the

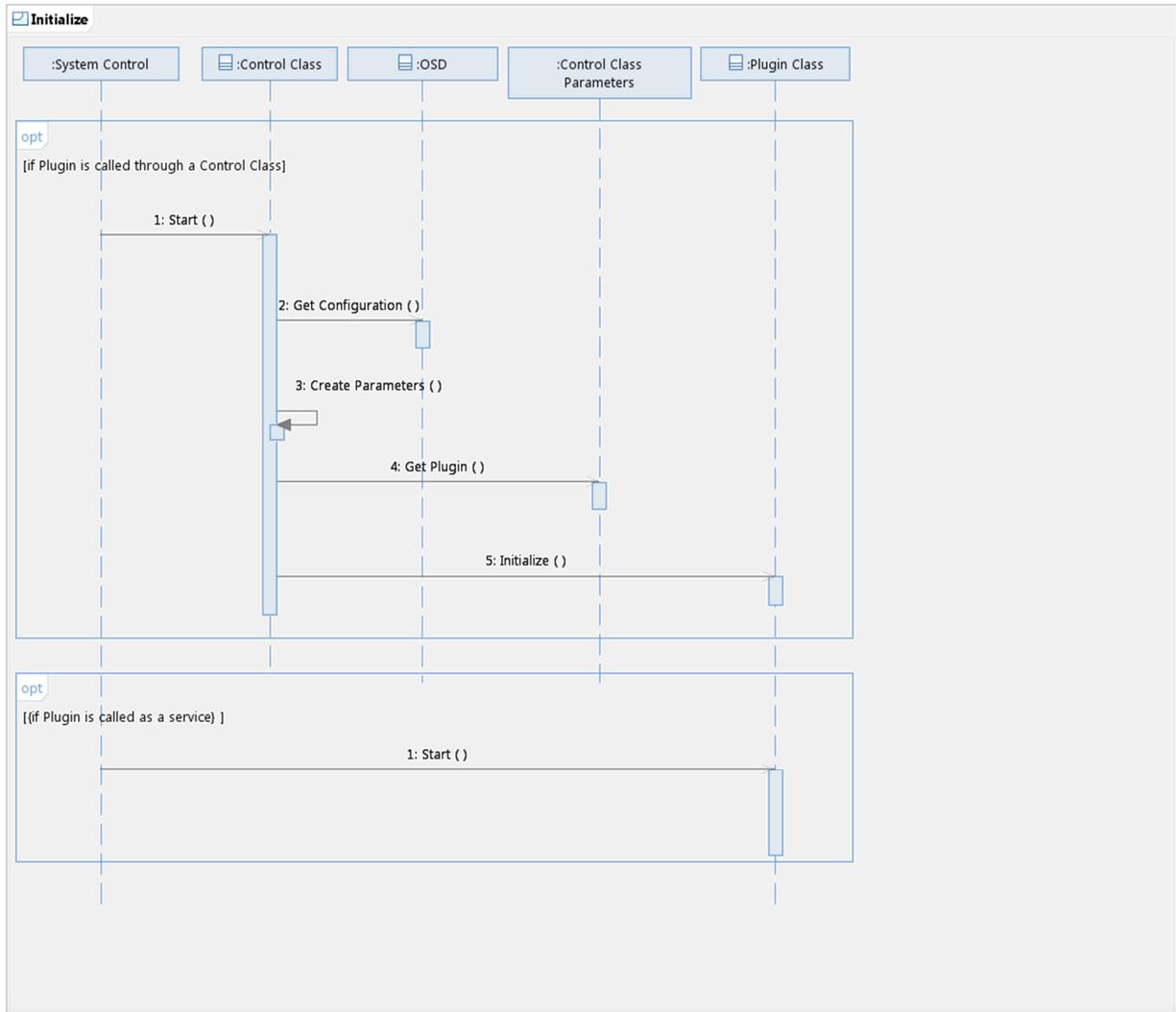
system uses these settings rather than the system default **Configuration** when invoking additional automatic processing to further refine the results. This ensures Analyst settings supercede default settings.

A further requirement is for the system to facilitate the update and replacement of algorithms. The architecture approach for this capability is to identify likely candidates for update and replacement and to model these candidates using **Plugin** classes. The **Plugin** class is invoked by a **Control** class through a **Plugin Interface** class. This isolates visibility of the **Plugin** class from the rest of the system. Figure 4-2 shows the relationship between **Control**, **Plugin Interface**, and **Plugin** classes. The **Plugin** class does not interact directly with other **Control** or **Display** classes in the system but operates on data provided via the **Plugin Interface** class. This limits requirements on the design of the **Plugin** class from implementing the control logic and data access logic developed for the entire system. **Control** classes can select among multiple **Plugin** classes that implement the same function by designating a configuration parameter to identify the desired **Plugin**.



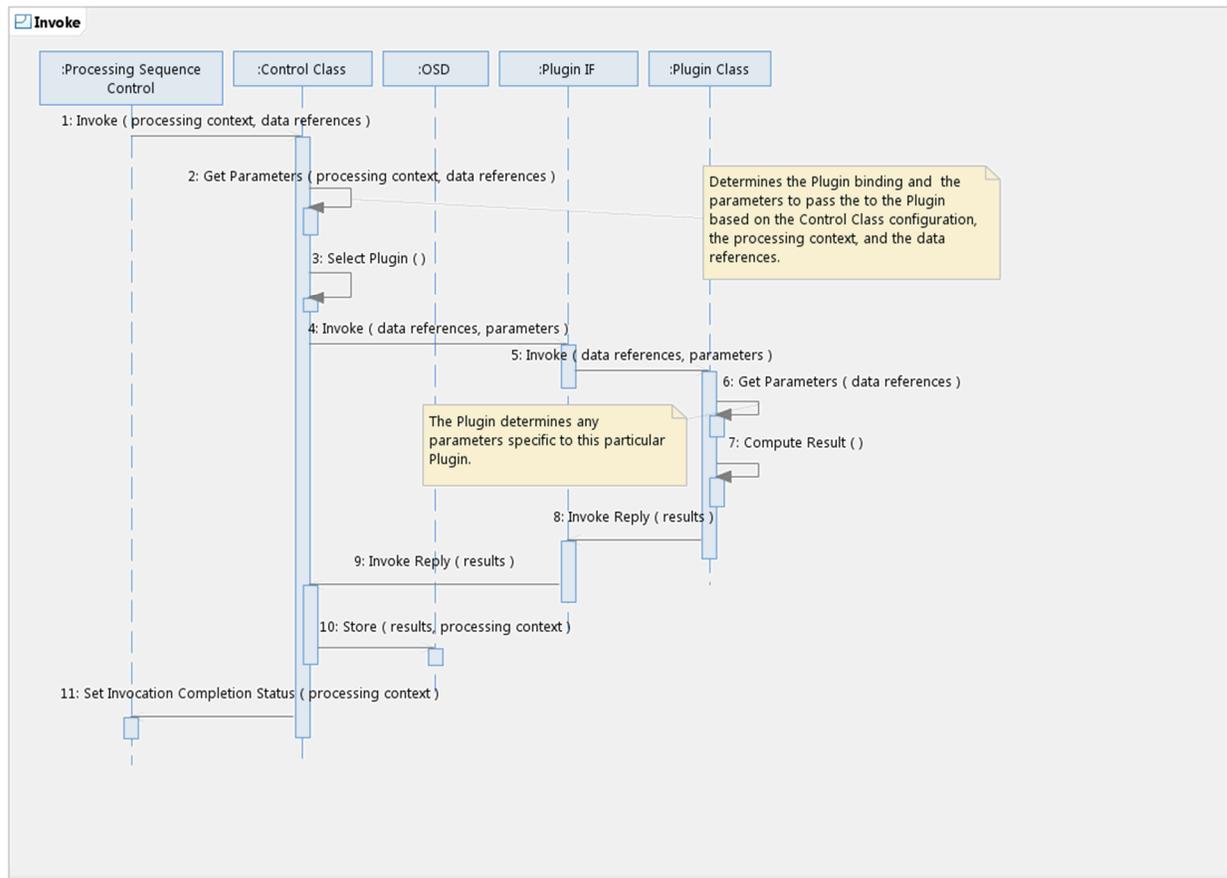
**Figure 4-2. Relationship between Control Class and Plugin Class**

The following sequence diagrams describe the pattern of interactions between Control classes and Plugin classes. This pattern is repeated in Use Case Realizations where Plugins are used. All the details of the interactions may not be repeated in the individual UCRs when they do not differ from the pattern described here.



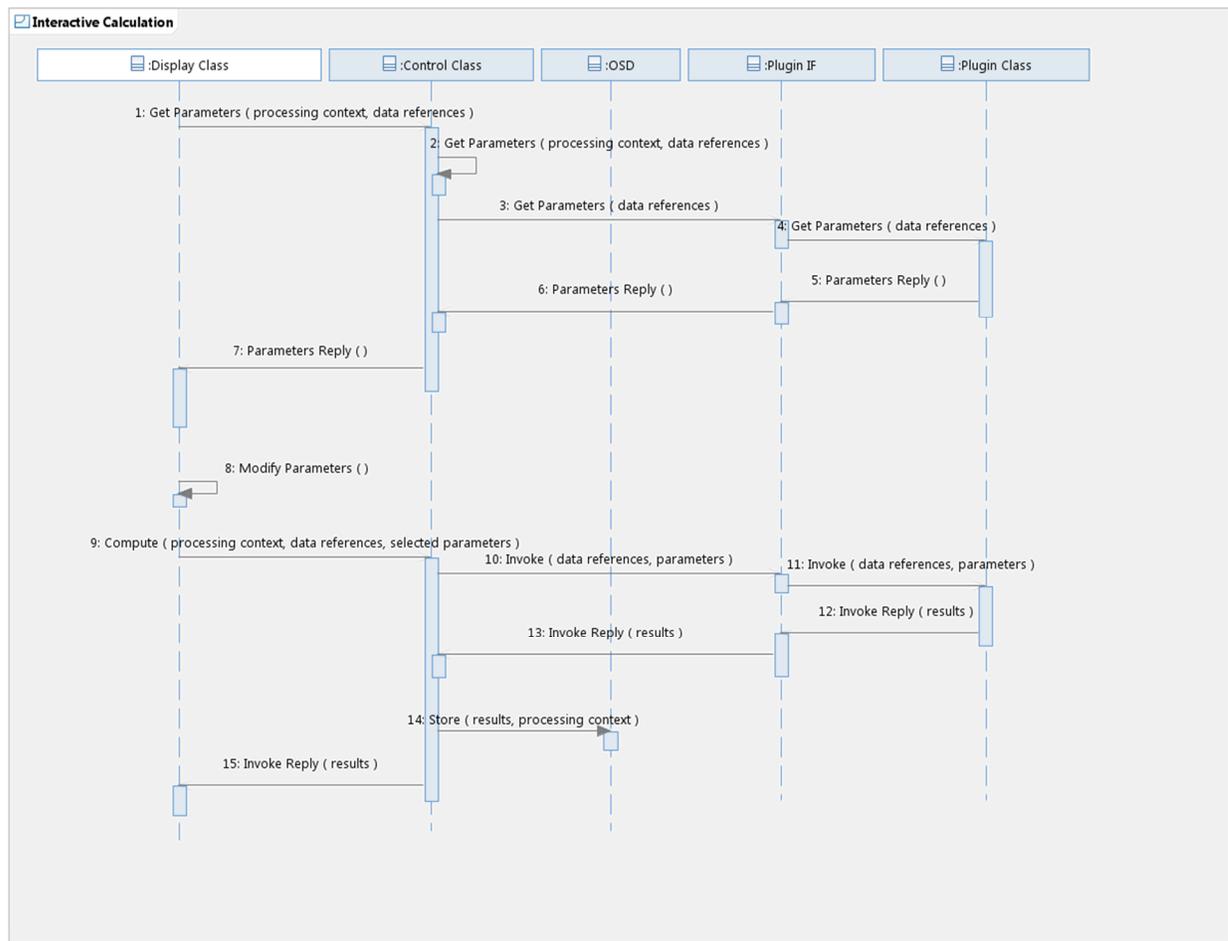
**Figure 4-3. Plugin Initialization**

This sequence diagram shows how Plugin classes are initialized. If the Plugin class is called by a Control class the Control classes initializes the Plugin classes when the System starts. The System Control mechanism calls each Control class to initialize. The Control class retrieves the default configuration from the Object Storage and Distribution Mechanism and creates the parameters that will be used to call the Plugin classes. The Control class also identifies the set of Plugin classes that implement the Control class functionality from the parameters and calls each Plugin class to initialize. If the Plugin is called as a service, the Plugin class is started by the System Control Mechanism.



**Figure 4-4. Plugin Invocation in Automatic Processing**

This flow describes how Control classes invoke Plugin classes to execute algorithms or implement models during automatic processing. The Processing Sequence Control Mechanism calls the Control Class when triggering criterion are satisfied with data references for the information to be processed and the processing context identifying the processing stage. The Control Class determines the Plugin Class to call and the parameters from the processing context and the data references. The Control Class calls `invoke` passing the data references and the parameters through the Plugin IF class to the Plugin Class. If the Plugin Class requires any plugin-unique parameters the Plugin Class determines those parameters from the data references. The Plugin Class then computes the result and passes the result back to the Control Class. The Control Class then stores the result based on the processing context and notifies the Process Sequence Mechanism of the completion status.



**Figure 4-5. Plugin Invocation in Interactive Processing**

This sequence diagram shows how Control Classes and their associated Plugin Classes are invoked interactively. The Display Class retrieves the default or last used parameters from the Control Class based on the processing context and the data references. The Control Class groups the parameters for the Control Class with the Plugin-unique parameters from the appropriate Plugin Class and returns the parameter set to the Display Class. The Display Class presents the parameter values to the user and the user modifies selected values and requests the System to compute based on the modified set of parameters. The Display Class calls the Control Class with the parameters and the Control Class invokes the appropriate Plugin Class passing the parameters similar to automatic processing. The result is passed back to the Display Class.

**Plugin** classes may create results containing information relevant to provenance, performance monitoring, tuning, etc. specific to the plugin implementation that go beyond what the **Plugin** is required to return by the **Plugin Interface**. Since the implementation specific results cannot be known a priori, each **Plugin** is responsible for defining **Entity** classes representing the plugin specific results and associating them with the results returned via the **Plugin Interface**.

Example **Plugin Interface** classes are shown in several UCRs, e.g., the System Refines Event Location, System Detects Signals, and System Builds Events using Signal Detections Use Case Realizations. Most of the **Plugins** perform calculations but some of the **Plugin** classes implement a **Plugin Interface** designed to provide data used in other calculations (e.g. the Earth Model Plugin provides values to Signal Feature Predictor Plugins). The table below lists the **Plugin Interface** classes that are currently in the Analysis Model. In some cases the Analysis Model also includes particular specializations of a **Plugin Interface** that may exist in the System to satisfy system requirements while in other cases the Analysis Model only includes the basic **Plugin Interface**.

Table 4-1. Analysis Model Plugins

Plugin Interface	Plugin Specializations	Defined in UCR	Used in UCR
Signal Detector		System Detects Signals	System Detects Signals, Detects Signals
Signal Onset Time Refiner	AIC Signal Onset Time Refiner, Waveform Cross-Correlation Signal Onset Time Refiner	System Detects Signals	System Detects Signals, Detects Signals
Signal Detection Associator	Match Signal Detection Template	System Builds Events using Signal Detections	System Builds Events using Signal Detections, Builds Event (UCR not modeled)
Waveform Correlation Event Detector		System Detects Events using Waveform Correlation	System Detects Events using Waveform Correlation, Builds Event(UCR not modeled)
Event Locator Plugin IF	Master Event Locator	System Refines Event Location	System Builds Events using Signal Detections, System Refines Event Location, Refines Event Location
Signal Feature Predictor		System Predicts Signal Features	System Predicts Signal Features, System Builds Events using Signal Detections, System Refines Event Location, Detects Signals (not modeled in UCR), System Refines Event Magnitude (UCR not modeled), Refines Event Magnitude(UCR not modeled), Monitors Mission Processing(UCR not modeled)

Earth Model		System Predicts Signal Features	System Predicts Signal Features, System Builds Events using Signal Detections, System Refines Event Location, Detects Signals (not modeled in UCR), System Refines Event Magnitude (UCR not modeled), Refines Event Magnitude(UCR not modeled), Monitors Mission Processing(UCR not modeled)
—END OF TABLE—			

Additional **Plugin Interface** classes will be created as UCR modeling progresses. The following table lists plugins that may be modeled in the future.

**Table 4-2. Analysis Model Plugins**

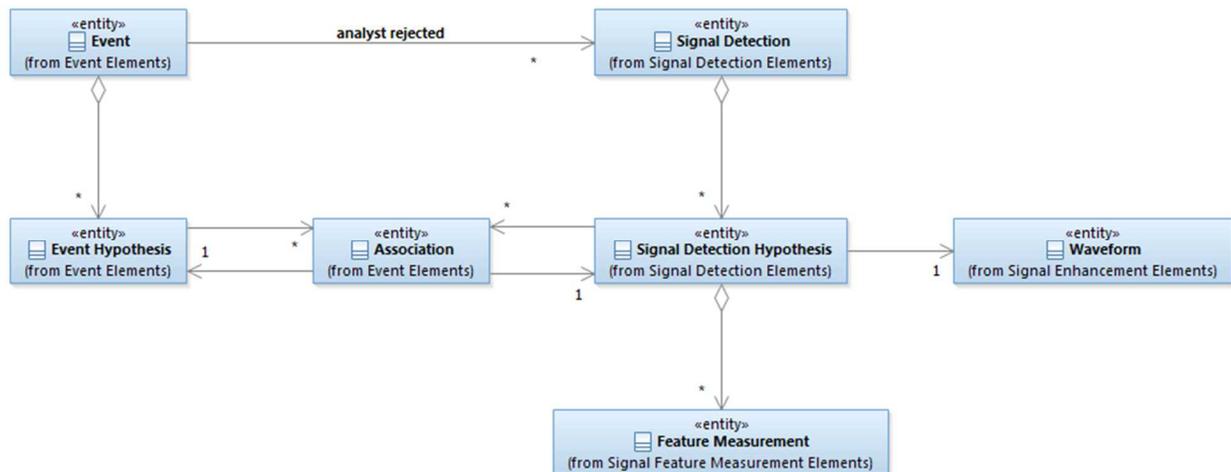
Plugin Interface	Plugin Specializations	Defined in UCR	Used in UCR
Multiple Event Locator		System Refines Event Location	System Refines Event Location, Performs Multiple Event Location
Atmospheric Model Builder		System Acquires Meteorological Data	System Acquires Meteorological Data
Waveform Quality Checker		System Determines Waveform Data Quality	System Determines Waveform Data Quality, Determines Waveform Data Quality
Signal Feature Measurer		System Measures Signal Features	System Measures Signal Features, Measures Signal Features
FK Feature Measurer		System Measures Signal Features	System Measures Signal Features, Measures Signal Features
Polarization Feature Measurer		System Measures Signal Features	System Measures Signal Features, Measures Signal Features
Waveform Filter		System Enhances Signals	System Enhances Signals, Enhances Signals
Waveform Rotator		System Enhances Signals	System Enhances Signals, Enhances Signals
Waveform Beamer		System Enhances Signals	System Enhances Signals, Enhances Signals
Moment Tensor Evaluator		System Evaluates Moment Tensor	System Evaluates Moment Tensor, Evaluates Moment Tensor
Magnitude Estimator	Network Magnitude Estimator, Station Magnitude Estimator, Relative Magnitude Estimator	System Refines Event Magnitude, Refines Event Magnitude	System Refines Event Magnitude, Refines Event Magnitude
Phase Labeler		System Measures Signal Features	System Measures Signal Features, Measures Signal Features
Event Conflict Resolver		System Resolves Event Conflicts	System Resolves Event Conflicts

Similar Event Finder		System Finds Similar Events	System Finds Similar Events, Compares Events
Analyst Performance Metric Calculator		Views Analyst Performance Metrics	Views Analyst Performance Metrics
Event Quality Calculator		System Builds Events using Signal Detections	System Builds Events using Signal Detections
Station Performance Calculator		System Builds Events using Signal Detections	System Builds Events using Signal Detections, System Refines Event Location
Network Performance Calculator		System Builds Events using Signal Detections	System Builds Events using Signal Detections
Event Comparer		Analyzes Mission Performance	Analyzes Mission Performance, Compares Events
Bulletin Comparer		Analyzes Mission Performance	Analyzes Mission Performance
Data Provenance Analyzer		Views Event History	Views Event History, Analyzes Research Events
—END OF TABLE—			

**4.2.7.4 Event Analysis Classes**

Events are the fundamental output of the System. The following diagram shows the relationships between Events, Signal Detections and Waveforms. An Event is the occurrence of some source of energy within the Earth's body, oceans, or atmosphere that can be detected by seismic, hydroacoustic, and/or infrasonic sensors. The Event class is composed of a set of Event Hypotheses each representing a different analysis of the Event. Each Event Hypothesis contains a summary of the contributing stations, associated signal detections, and a set of location solutions for the Event along with the parameters used to analyze the Event. One of the Event Hypotheses is the overall preferred version of the Event and represents the best available analysis of the Event. A Waveform is either the raw or derived output of seismic, hydroacoustic, and/or infrasonic sensors. A Signal Detection is a signal of interest. Similar to the relationship between Event and Event Hypothesis, each Signal Detection contains a set of Signal Detection Hypotheses representing different ways of analyzing the signal of interest. Each Signal Detection Hypotheses is described by a time interval on a Waveform. There may be signal enhancement techniques applied to the Waveform to help reveal the signal of interest. An Association object represents the relationship between an Event Hypothesis and a Signal Detection Hypothesis. An Event Hypothesis and Signal Detection Hypothesis that are associated to each other will each have a relationship to the Association class. If the Event Hypothesis and Signal Detection Hypothesis are later unassociated then their relationship to the Association class is removed. The Association class retains its relationships to Event Hypothesis and Signal Detection Hypothesis to track that the association existed in the system. Event Hypotheses and Signal Detection Hypotheses record the history of the analysis of Events and Signal Detections

and thus retain a significant portion of provenance in the system.



**Figure 4-6. Event Analysis Classes**

#### 4.2.7.4.1 Event Hierarchy

One of the significant features of the System is retention of intermediate results during event analysis. The previous system contained a limitation that only a single Event Hypothesis could be stored for each processing stage. The system will allow Analysts to store multiple Event Hypotheses during each processing stage as shown in the figure below. The Analyst will designate which Event Hypothesis is the preferred version for the Event for that processing stage. Another significant contributor to event analysis is the set of data available to each Analyst at the time they are reviewing the event. Late arriving data can modify the solution so a record of the data available at the time an Analyst is reviewing the Event is an important contributor to understanding the event solution result. The set of Event Hypotheses that compose an Event allows detailed post analysis of event formation process.

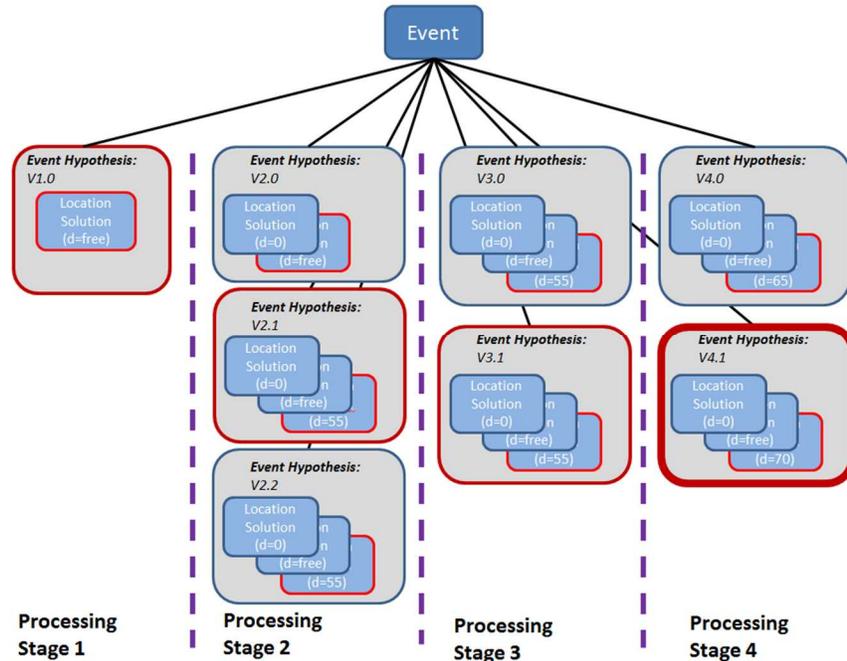


Figure 4-7. Event Hierarchy

#### 4.2.7.4.2 Processing Stages

Events progress through several stages of analysis both by the system and the Analysts. The figure below shows a notional sequence of processing stages in the current system. The ability to modify and define new processing sequences is a requirement for the System. To facilitate this goal the System will provide a method of defining processing stages and the processing sequences associated with the stage as part of the configuration of the system, The Processing Sequence Control Mechanism will control the processing sequence, calling the processing sequences associated with a processing stage. The operators will control the progression from one processing stage to another. This limits the dependencies between each stage to allow the sequence to be modified or updated with new stages or processing sequences

#### 4.2.8 Key Features

The System supports these new key features to enhance the effectiveness and usability of the new system: Undo-redo, provenance, replay, algorithm extensibility, remote ui, and Common Object Interface (COI).

##### 4.2.8.1 Undo-Redo

A required feature of the System is the ability for users to undo or redo commands during an analysis session. The system maintains a buffer of user-entered commands. The user may select to step backward or forward through the list of commands that the analyst has entered and the analysis returns to the state associated with the selected command. This requires the system to undo or redo both the Analyst entered commands as well as any automatic processing initiated by the system in response to the Analyst's commands. Each command in the buffer is responsible for undoing and redoing the operation represented in that command. Most commands will support undoing an operation by storing the series of changes leading

up to the analysis state just before the operation is executed. To undo the operation, the system replaces the current analysis state with the stored state. Other commands will support undoing an operation by executing the inverse operation. This type of undo does not have to use memory to store the analysis state but must ensure the undo operation returns the system to the same state it was in before the operation was originally executed.

The command buffer is cleared when the Analyst saves the analysis state. This occurs when the Analyst saves an event or saves the state of a scanning activity. Users have the capability to save multiple copies of an event to capture key points in the analysis. Users may open saved copies of events to review prior states or to continue analysis beginning from the prior saved state. The system does not save the command buffer with saved events so it is not possible for a user to open a saved event and then review the command buffer used when the event was originally analyzed. 4.2.7.4.1 Event Hierarchy and 4.2.8.2 Provenance describe how the system tracks event history and provenance.

---

#### 4.2.8.2 Provenance

Provenance is defined as data usage providing details regarding how the data has been used and modified and often includes information on how to cite the data source or sources. The system provides a more extensive set of provenance information allowing more detailed analysis and reconstruction of event analysis results. Expanded information includes records of when waveform data was available, which processing parameters were used to evaluate an event, and multiple saved versions of each event through each step of the analysis process. Provenance includes records of which **Entity** classes (e.g. Event, Event Hypothesis, Waveform, etc.) exist in the System, when those **Entity** classes are created, which user or process created those **Entity** classes, which prior **Entity** classes were used to derive the new **Entity** classes, and timing information indicating when **Entity** classes were created and when processing affecting those **Entity** classes executed. Much of provenance is captured by storing and querying relationships between data. Examples of these relationships and queries are storing and querying for all of the signal detections created by a particular Analyst during a particular time interval or finding all of the events created using a particular set of signal detections. The queries become more complicated as they traverse more relationships (e.g. finding all events rejected in one processing stage that were created in a previous processing stage using a signal detection created by a particular Analyst). The provenance design will address which provenance information is captured by the system, how it is persisted, and how it is queried when introspecting the system.

#### 4.2.8.3 Replay

The System provides the capability to capture waveform data and replay the data through the system to support algorithm development and testing. Also the System provides the capability to replay user commands. Waveform replay reproduces the time sequence of data arrival at the system. Waveform replay can recreate the situation where incomplete or corrupted data segments are corrected by later arriving data segments. Waveform replay also enables examination of processing late arriving data after events have been analyzed. Command replay allows replication of System response to user interaction with the system to allow more detailed error analysis. These capabilities are not available in the current system. The replay capability permits higher fidelity reproduction of the operational environment to support analysis and refinement of system performance.

Waveform replay can be implemented by injecting that data into the system through a test injection interface to emulate the external interfaces. A copy of all data received is re-sent to the system at the appropriate time. This approach requires saving the input data stream, copying the stream to the test injection interface, and sending the input stream to the remainder of the system.

An alternate approach records the reception time for each segment of data as the segments are stored. Then the test data injection process uses the reception times to forward data to the remainder of the system at the appropriate time to emulate processing the data as it was originally received. An advantage of this approach is the waveform replay does not rely on alternate external interfaces and all received data will potentially be available for replay without transferring the data to the test data injection process. However, because the test replay is embedded in the system separation of test and operational data will need to be enforced procedurally and/or through separation of test and operational results.

The most critical timing sequence to emulate for evaluating system performance is the arrival sequence of waveform data at the Data Processing partition. Therefore, waveform replay will be designed to inject data to the Data Processing partition in the time sequence of the original data.

There are some inherent limitations in reproducing an identical environment during testing even with the replay capability. Small timing differences of data availability during testing because of varied system loads or test platform configuration may affect the test results. Also, it is difficult to reproduce the entire state of the system, especially where algorithms use historic data in analyzing current inputs.

In spite of these limitations, the replay capability will greatly expand the ability to test the system.

#### **4.2.8.4 Algorithm Extensibility**

Extensibility of key algorithms is key for maintaining and improving the System. Algorithm extensibility allows the algorithms to be updated or replaced without affecting the remainder of the applications. Algorithm extensibility is modeled using **Plugin** classes as described in 4.2.4 Types of Analysis Classes and 4.2.7.3 Algorithms for Automatic and Interactive Processing.

#### **4.2.8.5 Remote User Interface**

An important consideration for the system is to provide a responsive user interface when the analyst is accessing the data remotely. The design of the user interface data communication needs to be optimized to minimize the delays introduced when the data is accessed remotely.

#### **4.2.8.6 COI**

The Common Object Interface (COI) is a definition of the software interface to the data on the System. The definition is distinct from the actual format of the stored data to permit the storage format to be optimized without affecting the application software. Persistence of stored information on the system is described in 4.2.6.3 Object Storage and Distribution Mechanism.

**4.3 Implementation View**

**4.4 Process View**

**4.5 Deployment View**

## 5 WORKS CITED

[1] Philippe Kruchten, "Architectural Blueprints—The '4+1' View Model of Software Architecture", Paper published in IEEE Software 12 (6), November 1995, pp. 42-50. Available: <https://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf> [Accessed 17 Dec 2015].

[2] G. Hohpe and B. Wolfe, Enterprise Integration Patterns, Boston: Addison-Wesley, 2003.

## 6 APPENDIX A. SPECIFICATIONS

The following System specifications shall be satisfied by the System Architecture:

- 2028 The System shall use a common object interface (data model and methods) for data.
- 2042 The System shall store automatic and interactive processing parameters in the database.
- 2043 The System shall store automatic and interactive processing results in the database for use by subsequent processing.
- 2218 The System shall make use of commercial off-the-shelf (COTS) and open source software where possible.
- 2219 The System shall use commercial off-the-shelf (COTS) and open source software with a defined upgrade path.
- 2220 The System software shall be written using a minimum number of programming languages.
- 2224 The System shall implement dates and times that include leap years and seconds.
- 2226 The System shall use year 2038 epoch rollover compliant date formats.
- 2233 The System software shall be maintained and controlled via configuration management software.
- 2262 The SUS/TST Subsystem shall be a functionally redundant copy of the OPS Subsystem.
- 2317 The System shall maintain a mission profile operating 52 weeks a year, 7 days per week, and 24 hours a day.
- 2322 The Training Subsystem shall support mission duration of eight hours per day, five days per week, 250 days per year.
- 2331 The System shall store on the System all existing data and five (5) additional years of data.
- 2332 The Training Subsystem shall provide storage with sufficient capacity to accommodate thirty (30) days of multi-phenomenology waveform data for stations available on the OPS Subsystem.
- 5688 The System shall provide the System User the capability to remotely access required user interface functions on the OPS Subsystem from a remote connection over a secure connection.
- 5689 The System shall provide access to all Analyst capabilities from a remote location over a secure connection.
- 5725 The System shall use date formats with four digit years.
- 5731 The ALT Subsystem shall be a copy of the OPS Subsystem in software and hardware not physically collocated with OPS.
- 5738 The System shall reuse suitable existing software where practical.
- 5739 The System shall use open-source software whenever possible.
- 5740 The System shall use open-source software when both open-source and commercial software are available.
- 5766 The System shall support at least 1000 Authorized External Users.
- 5767 The System shall support each Authorized External User requesting up to 4GB of data per day.
- 5768 The System shall support at least 30000 requests for data and products per day.
- 5831 The System shall use relational database management systems that support ACID transactions, referential integrity and fine grained locking.
- 5832 The System shall use a distributable open source database for Standalone Subsystems.

This is the last page of the document.