

THE MOULAGE SYSTEM

(Making The Right Impression)

By: **Scottie Swenson, Yvan Rochon Ph.D. and Lucy Akolzina**

1.0 Executive Summary:

The Moulage System is a real time data sharing & analysis system framework. It provides the application foundation to allow investigators to import data from virtually any source, perform real time analysis of the data in question, and query other available data sources (networked database systems). The data can then be exported to a colleague (including the analysis software needed to view the data), to a formatted report, or to a binary data file. The Moulage System can also provide access to high performance data systems that allow large computational analysis of data at speeds normally inaccessible to the Principal Investigators (PIs).

The Moulage System has been built exclusively in the Java language from Sun Microsystems, Inc. The Java (Java is trademarked Sun Microsystems, Inc. see "Copyrights:" on page 14) language is a "compile once, run anywhere" programming solution that has been embraced by all major and most minor computer operating system (OS) manufacturers. By constructing the entire system in Java, the Cellworks Project (CWP) has provided a highly transportable foundation for any scientific application that has the added benefit of an automatic and transparent software distribution capability (i.e. software updates can be added automatically via the network with out the need for user intervention).

2.0 Abbreviations used:

API	Application Programming Interface
CWP	Cellworks Project
GUI	Graphical User Interface
JDBC	Java DataBase Connection
JVM	Java Virtual Machine
IO	Input and Output
OS	Operating System
PI	Principal Investigator
URL	Universal Resource Locator (e.g. http://cellworks.washington.edu)

3.0 System Description:

The Moulage System is not an application itself, rather it is an application foundation. The CWP programmers have taken all of the "normal" application methods and functions which are needed to write a scientific application, and abstracted the common elements into a set of foundation programs. Applications can be rapidly developed using the Moulage System as the foundation since the work to develop methods or functions to handle data control, debugging, file input and output, applet controls, applet parameter support, multi-threaded event handling, and many more (see the Moulage class API documentation for more details) has already been done and is ready to use. Thus, new file formats can be supported in a matter of hours instead of days, and entirely new analysis programs can be developed in weeks instead of months. In addition, all programs developed using the Moulage System gain a common application programming interface (API) that allows the analysis code and applications to be tied together rapidly.

The Moulage System is multi-threaded for parallel, multi-process or multi-host operations. The design allows the ability to perform multiple operations (such as data loading and data mining) in parallel. In order to perform multiple operations simultaneously, the entire system is capable of detecting and preventing failures due to multiple processes attempting to update a specific resource at the same time. Multi-threaded design is implemented from the

outset and is a consideration in any updates. This design enables the Moulage System to take full advantage of modern multi-process platforms and parallel operating systems.

3.1 Moulage System Design Goals:

- 1) Transparent access. Allow users to access a wide variety of data and analytical methods from local or remote host computers.
- 2) Accept multiple data sources. Allow importing and exporting from any source data, no matter what format it is in.
- 3) Allow easy sharing of data. Provide an environment to individuals or collaborating groups that includes data mining, email, newsgroups, and other communication technologies.
- 4) Keep data in form usable by object-oriented databases. Due to the nature of the most commonly analyzed data types an object based approach is needed to allow for the diverse types of data. Since the datum needs to be in an object form to analyze, storing it in that form is the best approach both for speed of retrieval and data validation.
- 5) Provide multiple tools for analysis. Since various data types can be analyzed with different applications, the system must ensure that the applications are compatible with each other.
- 6) Dynamic version distribution. Ensures that the user always has the most current version of the software without requiring the user to search for and install each new version.

3.2 Moulage System Base Components:

Since Moulage is based on the Java programming language, great care has been taken in the design to ensure that the system adheres to object-oriented design patterns, and permits representation of extremely complex data. The object-oriented design pattern also causes the Moulage System to be fully modular. Each component performs a small subset of functions that are specific to that component's type only. Most of the inter-object interfaces and functions have been built into each component at the Moulage class level.

The design allows any Java programmer to implement a new application or supporting class to the overall system simply by extending the appropriate class type and implementing the few required methods. Creating a whole new analysis application is done simply by extending (inheriting) the Moulage Inspector class and adding in only the actual user interface and analysis methods. Loading of source data, network communications, database interactions, and universal user interface components (i.e. open new file, add data set, view textual source data) are already written and functional immediately. To build a new analysis environment for analyzing new data types requires the additional step of defining the new data type(s). To create new data a type a programmer merely extends the Moulage Imprint (the object representation of a specific data type) adding the specific parameters needed to define the new data type.

Moulage System components, shown in Figure 3.2.1, "Moulage Design Structure," on page 3, can be enhanced or replaced at the Moulage class level without affecting current systems. By carefully defining the API, all of the underlying methods can be enhanced without introducing problems to any currently existing applications. By the use of dynamic class loading across the Internet these enhancements would immediately be integrated into all subclasses transparently to the scientists, aside from the new functionality or speed increase or bug fix that was changed. In other words the underlying foundation can be repaired and enhanced without breaking or requiring rebuilding the applications that were built on it.

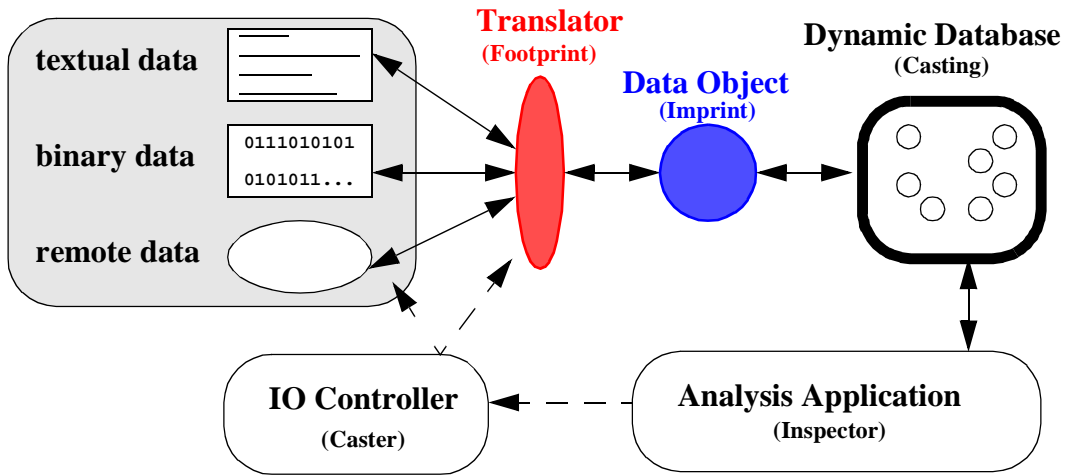


Figure 3.2.1 Moulage Design Structure

The design is meant to package all of the routine functions like input/output (IO) into classes which need not be looked at further than using a limited API set of functions to request specific activities. Thus the design effort shifts to dealing with the key issues as shown in Figure 3.2.2, “Moulage Component Layer Diagram,” on page 3. These areas are the only ones that need to be addressed in the creation of any new application. The Moulage System provides the framework shown in Figure 3.2.3, “Network & API Layer Diagram,” on page 4.

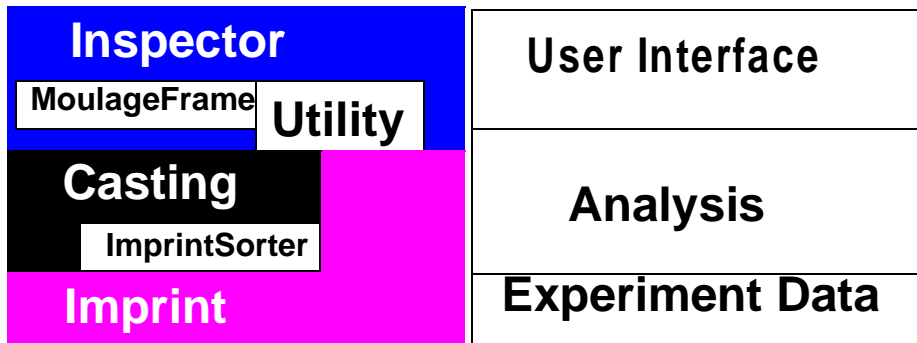


Figure 3.2.2 Moulage Component Layer Diagram

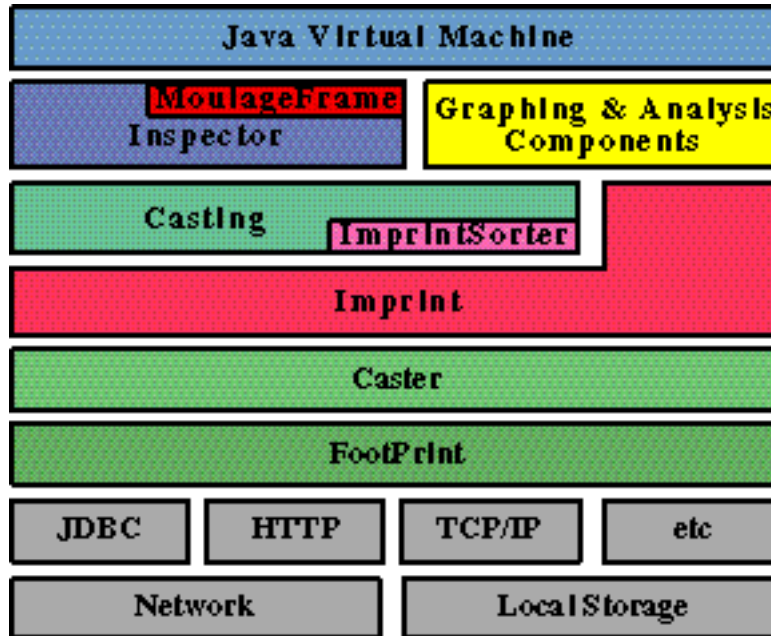


Figure 3.2.3 Network & API Layer Diagram

3.2.1 Inspectors:

An Inspector is the foundation for a tool that performs a specific set of analysis functions against one or more sets of data, either after data has been collected or in real time. An Inspector may or may not have a graphic user interface (GUI). Inspectors are the keystone class for the Moulage System providing both the user interface (automated, command line, and/or GUI) and the analysis of data. The Inspector class furnishes generic option controls and interfaces needed to perform data analysis. The API follows the Moulage System’s primary design goal of dynamically linking all necessary routines for any data type. This allows for a completely heterogeneous environment that is well suited to analyzing multiple data types simultaneously on one screen facilitating collaboration, presentation, and data mining.

Inspectors are designed to work via a command line interface, icon (or script file) launches, through web browsers as applets and through interface calls from other Java classes. Input parameters are standardized and recommendations are provided to allow additional parameters so as to maintain a common interface logic and API. For example, when a Moulage Inspector is called through a web browser as an applet the logic for setting up the internal references (like Debug, FileBase, etc.) are already handled within the base class. If an Inspector is launched via a browser it automatically looks for the applet parameters: “Debug” which can be any of “ON”, “OFF”, “true”, or “false”, not present means OFF; “Filter Index” which is a whitespace separated list of the index files to use for this instance; “Input Files” which is also a whitespace separated list of the source data files to load; and more. File or data references can be URLs, direct file references, or pointers from the current applet’s “FileBase”.

The Inspector class’ API provides an interface which can communicate with other Java or Moulage classes in the same Java Virtual Machine (JVM) or through network connected JVMs. The communications interface can be used to start additional data analysis by passing object references between the classes. Thus Inspectors can be easily tied together or to other analysis tools that work on the same types of data.

Moulage components built by the CWP have established a precedence for using this cross application communications for all start-up methods. It is a central design element of the Inspector class that no matter which way the class is started one method gets called to start up the application. Thus Moulage based applications can also be fully operational applets with very minimal effort. Hence, there is only one start-up route no matter what application start-up method is used. This allows the Inspectors to easily deal with multiple directions of start up (i.e. from another Java class or Inspector, as an applet, or from the command line).

3.2.2 Caster:

The Caster is a dynamic source identification and import/export class. This class contains the majority of IO logic for the Moulage System. The Caster can read source files, index files of keys to Footprints (described later), and a data objects to hand off to the Footprint(s) from a number of different methods (i.e. local file, directory scans, pattern matching, network URLs, etc.). It analyzes all of the sources to determine their type by using an internal set of logical rules against index data. Once a source has been identified it is either: converted internally and directly inserted into the mini real time object oriented database management system (Casting class described later) for analysis; or the Caster dynamically locates and loads a Footprint to perform the data translation from the source data into the Casting for analysis.

The Caster is designed to be a completely transparent program to the programmers and users. It is started with a simple queued request and uses the JDK 1.1 event model to inform Java 1.1 event listeners of its progress. It performs data loading, source analysis, dynamic class locating and other functions completely independent of other operations underway. The Caster does this by using a number of multi-threaded techniques.

The CWP is dedicated to insuring this class supports any needed IO methods except for database (JDBC) look ups. The reason database connectivity was left out is that all sites have a unique database connectivity model that supports differing styles of queries. It was decided that specific database needs would be left to the site support programmers and be done through the Footprint classes. Thus allowing a URL to a database which would trigger a hand off to a Footprint class for the actual interactions with the database.

3.2.3 Footprint:

A Footprint is a small independent data factory responsible for translating data from one specific data input format to a specific object or objects representation of the data and vice versa. The Footprint class contains the normal interface controls for communicating with the other Moulage classes. All of the source data handling, type checking, reference control, etc. is handled automatically.

A Footprint is needed to perform the import and export functions for each data type. Collaborating sites that wish to use Moulage classes on their unique or proprietary data need merely to create a new Footprint on their host(s) to import and export their data. Footprints can be public or private and internal data types can be mixed with external data without compromising proprietary files or data.

Creating a new Footprint is achieved by extending the base Moulage Footprint class and writing three simple methods:

- 1) a short initialization method to describe the Footprint to the executing Moulage components;
- 2) an import method which only needs to translate from an already loaded file to an already type checked Casting instance; and
- 3) an export method to translate from an Imprint (or Imprints) to an output byte stream for exporting the data back into the original data file format.

The initialization method is a very small method called "*andMore*". This method is a constructor method for setting up the needed variables:

- 1) "version" is recommended to be modified by placing a class name, version data, and copyright statement onto the front end of this string;
- 2) "thisClass" is a full class name identifying this extension's name which is used for cross references with other components and for debugging output; and
- 3) "FootprintFor" is required to be defined to a string which contains the full name of the **Imprint** that the implementation is designed to write to.

The import method is called "*makeCast()*". It must have the code to read data from a loaded Java **ByteArrayInputStream** (a.k.a. the source file) and translate it to the appropriate **Imprint**. This method must also make sure to add the translated Imprint(s) into a **Casting** instance which will be passed to *makeCast()* by reference.

The export method is called "*imprintCast()*". It needs to process a loaded array of **Imprints** into one or more output files formatted to be acceptable source data. These output files will be placed into a Java

ByteArrayOutputStream(s) in an array returned to the calling method. This method enables the export of any Moulage data to a local format for use in programs external to the Moulage System.

The Moulage Footprint class has the ability to receive data source file(s) as: a preloaded **ByteArrayInputStream**, in which case there is only ONE file to process; a string of file names (URLs, direct path or some appropriate combination) as whitespace separated names; or an Object reference which will be checked against the expected **Casting** type (defined in the "FootprintFor" string) and added into the appropriate **Casting** instance if it is appropriate.

The Footprint class is designed to receive the destination reference if only one type of data is being loaded. However, the reference first passed to the Footprint should normally be an **Inspector** (this is also checked). If it is an **Inspector**, a call to the back channel method ("*needPlaster()*") is initiated to ask for the appropriate **Casting** type. Once the first call for a **Casting** reference is performed, the reference is type checked again. If it is bad (null) or of the wrong type, the Footprint fails out with a call to debugging logging method ("*tattle()*") and exits nicely, returning control back to the calling object.

3.2.4 Imprint:

An Imprint is a specifically designed data type object (e.g. an amino acid sequence, a "DNA sequence", etc.). This class is individually designed for each type of data it represents and holds data relevant to a single instance of its type (e.g. 1 DNA sequence, 1 protein sequence, 1 gene, etc.).

Each Imprint has two special methods that are the validation methods for confirming that the Imprint holds reasonably good data. Both of these methods must return a boolean value (true or false) that indicates if the instance is a valid data representation of the type of data the Imprint was designed to reference. While performing these checks, the methods are expected to use calls to the debugging method "*tattle*" to describe the check progress (especially a failed check with the reason for the failure). This allows for validation and return messages to be passed to the user or data management system that can later be used for cross checking or debugging purposes.

The first of these methods is called "*perstringe()*." The *perstringe()* method performs all quick surface level (high speed) checks that can be performed in real time without adversely affecting run time. This method is called by an instance of **Casting** before it is added into the internal list of Imprints.

The second method is called "*battue()*." The *battue()* method performs all deep data mining validity checks that can be performed without concern for run time. The *battue* method is meant to be called only from inside of the large OODBMS repository. This method, by definition, can take considerable time to complete. This is the primary means by which the automated experiment revalidation will be implemented within the CWP's OODBMS system.

3.2.5 ImprintSorter:

An ImprintSorter provides the correct methods for comparing one object to another based on some type of method(s). Additionally, this method is to be used for locating a specific data object within a **Casting**.

An ImprintSorter must be named after the Imprint for which it was made. The name must follow the syntax of **Imprint class name** + "**Sorter**". This standard is used so that a specific sorter can be located and loaded at setup time for a **Casting** via a call to the Java classloader in the form of `Class.forName(Imprint.thisClass + "Sorter")`.

The basis for the ImprintSorter comparison method is laid out such that for any type of data there must be a means to order it for visualization and reporting purposes. For data types where this is not an option, no ImprintSorter need be defined. It will not cause an error if there is no associated ImprintSorter. All comparisons are of two elements at position A and B: a true should be returned if item A is less than (<) B; a false should be returned if A is equal to or greater than (>=) B; all sorts are on the basis of ordering the lists from smallest to largest quantities. You may reverse that method by changing the return values here. In any case if A is equal to (=) B a false should always be returned to prevent un-needed swapping of equal elements.

The considerations for the comparison logic are:

- 1) The 'A' (first) Element is always considered to be closer to the "start" of the sorted list of data elements.
- 2) If the two elements compared are the "same" value, return a false.
- 3) Return a false if the comparison cannot be completed.

3.2.6 Casting:

A Casting is a generic, independent object manager for **Imprints** and their related **ImprintSorters**. It provides a complete OODBMS style API used by the other Moulage components as well as functioning exactly like a vector. In fact the Casting class can be directly integrated into a class by a direct substitution for **java.util.Vector**.

For each data type to be analyzed an **Inspector** should create and initialize a Casting instance. Once a Casting has been given an **Imprint** it adjusts its internal methods to handle only other **Imprints** of the same type. A Casting instance can be recycled by clearing it of all **Imprints** and handing it a new **Imprint** of a different (or the same) type.

All calls for sorting, enumeration, and direct random access to the data elements are handled by this class. All returned data is either stored in an external public references for this class (as defined for each specific data type) or placed into an external **Imprint** object created for just that purpose.

Castings are set up to be sequentially safe. They treat all internal data as immutable. If an object is retrieved from a Casting, what is actually passed to the **Inspector** is a clone of the internal **Imprint**. The internal element can be changed by overwriting it with another **Imprint**, but note that this is a deliberate action. This also provides an automatic “undo” type function to the **Inspectors** since they are only working with copies of the data.

The Casting class is as thread safe as possible. However, there still exists the possibility that an **Imprint** object can be used by the Inspector that wrote the object in after the **Imprint** is added into a Casting. Some discussion has occurred that suggests the Casting should also replicate all inbound Imprints and store the cloned copy. This feature may be added to version 3.0 of the Moulage System. Until then all methods must be careful to dereference Imprints that are placed into a Casting after the insert operation. Data extractions are not affected and Moulage has a built in safety mechanism for validating all input data objects within each Imprint that will at least detect “improper” data.

3.2.7 Utility:

The Moulage Utility class is a small part of Imprint and Inspector. It has the Imprint’s error handling routines and the Inspector’s event and execution methods. This class has no predefined interface other than those specified in the standard Moulage API. This allows for small bean like utility classes to be built while retaining the Moulage basic API for debugging and some Imprint features such as the error message collection. In essence this class is meant to handle data analysis or systems interactions specific to some application or site requirement without user interface overhead.

3.2.8 MoulageEventMulticaster:

This class provides an event publisher for ANY moulage based application. The design is as parsimonious as possible. However, every single Moulage event adapter operates in it’s OWN thread. Therefore this class is written to be completely thread-safe. The publishing mechanism is a functioning multi-cast event publisher that is wrapped into an immutable structure consisting of a chain of event subscribers and will publish events to those subscribers’ threads. Since the structure is immutable, it is safe to add and remove subscribers during the process of an event publication operation.

The immutable chain design is from the Numerical Recipes in C book. The class design is modeled after the Java AWT EventMulticaster class written by Amy Fowler and John Rose in 1997 (a beautifully designed class!). The API for this class was purposefully designed to mimic the EventMulticaster style so that it would be comfortable to use for any designer who is also working with the Java 1.1+ AWT and Beans APIs.

3.2.9 Ensemble:

This class is a collection of the base Moulage component methods. It exists because multi-inheritance would have been handy to implement these methods across all of the Moulage components. However, version 1.0 of Moulage used the inheritance model and, although successful, the model left a lot of functionality out.

So version 2.0 Moulage broke the huge inheritance lines and each class kept a reference to an interface that referenced all of the original methods. However, as expected originally with such a model, maintenance across the classes became a problem. So this class was created to handle the 'base' operations and each Moulage component was then free to forward the data to this class to handle the normal expected actions before handling any specific extensions.

It should not be necessary to call directly for Moulage implementations as a call to `super.method(args)` will get to this class if and as needed.

4.0 System Operation:

All entrances to Moulage System based applications are normally through an Inspector (the analysis interface for both real-time GUI interaction and automated analysis routines) or perhaps a unique Utility class. The starting class is given none, one, or more optional inputs. The initial input is normally a list of source files and a list of index files used to load the data, debugging status and possible operational parameters. If the Inspector is running as a real time user interface, these items may be dynamically modifiable by the user during execution, depending on the application design. This section along with Figure 4.0.1, "Data Acquisition," on page 8 will explain the general interactions between the Moulage classes for a generic implementation.

Each data source reference is passed through the Caster, which identifies the file type and the associated Footprint class (i.e. the translator or data filter). The Caster then attempts many approaches, both locally and via network accesses to dynamically load the needed Footprint. If it fails it reports the failure and moves on; if it succeeds, it constructs the Footprint instance, setting thread priorities and debugging state. Then, the Caster passes to the new Footprint instance the source data and instance references. The Footprint breaks off into its own thread and begins loading the data. The Footprint(s) filters the data source(s) into the needed Imprint Class(es) and adds the new object(s) into the specified Casting object(s). Once the Footprint is running the Caster then begins work on the next source data. This continues until all sources have been passed to an appropriate Footprint or were unable to be loaded. The Caster then waits for all Footprints to finish and then it calls back to the Inspector notifying it that the load is complete.

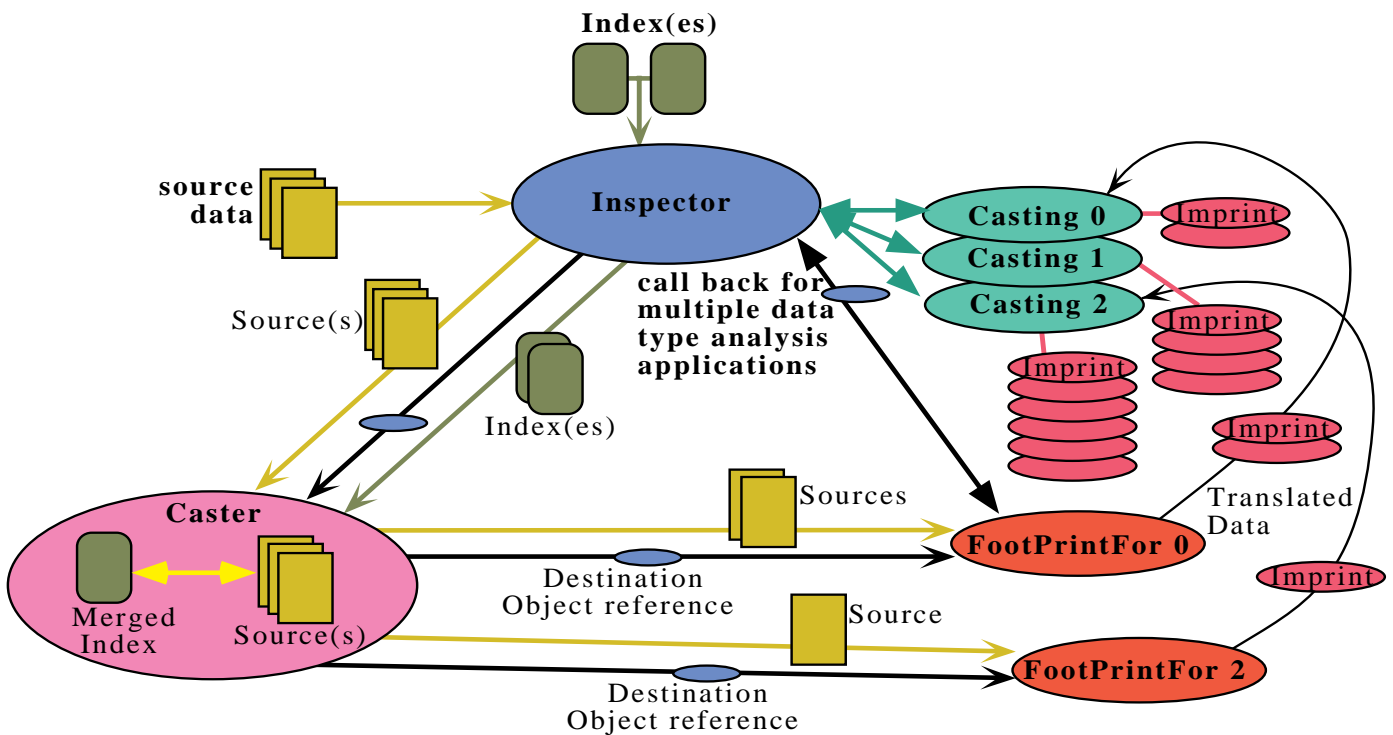


Figure 4.0.1 Data Acquisition

The Caster has a built in governor method that determines CPU load, operational impact (system load) and keeps track of the number of subprocesses running. The governor can be set to use any and all of the tracked values with upper bound limits to keep the processes from overloading the computer or the number of active connections to a database system. If the loads are such that another process can be started the Caster moves on to the next data object.

If any of the loads are over the threshold the Caster waits for the processes or resources to become available to continue. So the Caster can run single threaded or dynamically.

The Inspector application does not need to wait for the Caster to finish before moving on to other tasks. In fact, if a user wishes to add in other data sets via the GUI, additional requests are queued to the Caster which will handle the additional data as it can.

As each Imprint is created by the Footprints and added into the Casting instance(s) the data is spot checked by the Casting before the data is added into its list. One design criteria for the Moulage data structure is to provide high data integrity. Data integrity means that no data within the system has been accidentally or maliciously altered. To ensure data integrity, the Imprints should be designed to assist in determining if data has been altered to “look” valid without being valid. Data integrity also means that only the individuals who have been explicitly given authority over the data objects can add, alter, or remove the stored information. In building the Moulage System scheme, the CWP has provided a data object class that can not be altered except through its interfaces. This controlled interaction with a Moulage Casting based data object allows programmers to build into the object any integrity and security checks that are needed to provide a high level of trust in the data itself.

In the future the Moulage database operations will provide additional integrity checks. That is, as more data of a specific data type is entered, internal user independent criteria will be established to compare elements for consistency. The criteria are derived by parsing an element of a particular data entry and comparing it against itself as represented in some other data structure. For example, within the Imprint for mass spectrum data, the sequence of the peptide resides within a file containing the sequence of the protein in which the peptide region is represented. As each entry to the Visualize Mass Spectrum Inspector occurs, a separate method within the Moulage Imprint is called to check the peptide sequence with the protein sequence file and confirm a match. If the match is confirmed, the object is loaded without alteration; if the match does not occur, an alert [a visual indicator “*”] will be inserted at the beginning of the sequence entry. Additionally, when data is inserted into our data archives, the retrieved data structures will perform intensive data checks to validate all entries.

For all Moulage based applications the source data passes through one or more Footprints and is added into one or more Moulage Castings during initialization. The source data can include data already in a Moulage Casting format coming from a larger archives or database. Every time the data is loaded, the Moulage System performs consistency checks to insure data integrity.

After all data sources have been parsed into appropriate Moulage Imprints and inserted into one or more Castings the Inspector application has a mini OODBMS with partially validated data. These small, self-maintaining databases allow various Inspectors and data analysis tools (available across the Internet or intranets) the capability of providing real time visualization, document composition, data analysis, information exchange, and/or database insertion. See Figure 4.0.2, “Interface Relationship Post Acquisition,” on page 10 for the visual representation of these class interactions.

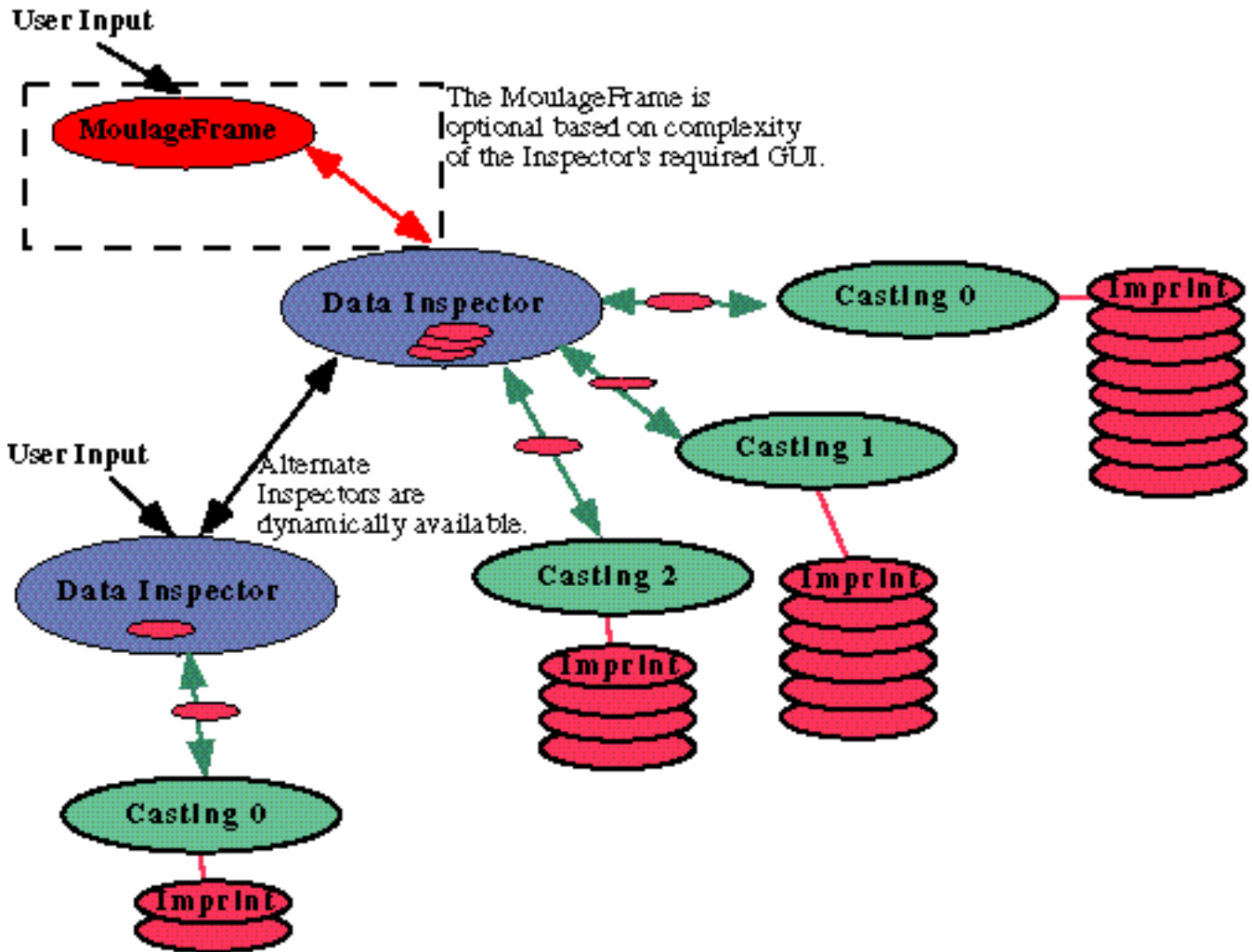


Figure 4.0.2 Interface Relationship Post Acquisition

5.0 Functional & Under-development Inspectors:

This section provides short descriptions and some details on the currently functional or in-development applications based on the Moulage System. It must be emphasized that this list is considered incomplete, as the Moulage System may be extended and used by any lab with access to the system classes without the CWP's knowledge. We freely provide access to the APIs that allow other programmers to extend the various Moulage classes. All extensions of the Moulage System have access to all of the globally available features without breaching the security of any collaborator's experimental data. Additionally, as the system is expanded, any extension that references our site's classes will receive all the new features and functions without ever having to modify or recompile their extensions. The Moulage System is entirely reusable and backwards compatible.

5.1 Data Manager:

The Data Manager Inspector is a Moulage System Component that was added to the system to manage other Inspectors. It accepts data from source files that describe available data sources, related index file sources, related Inspectors, a short title, and a long description. It provides a GUI listing of available data and the associated Inspectors in a central control panel. It also communicates with currently running Moulage components and can control their operations. All Moulage process operation components (Inspectors, Footprints, Caster) report to this

Inspector if it is operating. The individual processes are displayed on a list window panel allowing the user to observe the systems activities.

Currently, this Inspector is used to provide a web resource GUI listing of all available data within a collaborators web site. The CWP is using this Inspector to separate collaborating labs data sources into independent locations or sub-domain web pages.

The Data Manager is capable of shutting down one, some, or all currently running Inspectors from its control panel. The ability to manipulate Moulage Components thread priorities was added into Moulage as of version 2.1. This Inspector is slated to allow user manipulation of these priority threads from the control panel in order to permit dynamic reassigning of process priorities by the users.

5.2 Thread Snooper:

The Thread Snooper Inspector is a unique implementation of an Inspector. It does not accept data from a remote source. Instead, it uses the current Java Virtual Machine as the source of its data. This Inspector is actually a debugging tool that displays all currently running Java process threads. This was constructed as a Moulage Inspector as a demonstration of the generic application abilities of the main system design. It is being enhanced to allow for thread priority manipulation under the Moulage version 2.1 enhancements.

5.3 Applet Window Control:

The Applet Window Control Inspector was designed to allow easy access to any applet. All Java applets are specified to be accessed and started in a certain preset method. This Inspector is designed to be an integral Moulage component that will allow any applet to be run within the Moulage System. It accepts one or more applets which are then started in independent windows with all of the expected applet interfaces in place.

Eventually this Inspector is intended to allow the easy construction of other Inspectors. A programmer can target the individual Inspectors specific to an applet environment and then use the Applet Window Control Inspector to provide for independent execution via the command line or Data Manager.

5.4 Consensus

The Consensus Inspector is being constructed by a collaborating lab at the University of Colorado under the Direction of Dr. Gary Stormo. This Inspector is intended to provide a Java application version of Dr. Stormo's consensus sequence analysis application.

The consensus application performs complex analysis on one or more sequences (DNA or protein) to locate unspecified pattern similarities within the input sequences. This system provides a large number of analysis options and can be used to perform analysis on database retrieved sequences.

By converting the entire application to Java and integrating it into a Moulage Inspector, Dr. Stormo intends to allow for easy access to the methods as well as allowing it to be tied into alternate Inspectors which have the ability to work with sequence data.

5.5 Detente:

The Detente Inspector was built by the CWP during the early stages of the Moulage System construction. This Inspector manages external data sources, allowed query construction and response controls. Each data source, by definition, will have one associated Footprint (this includes data base systems). The CWP is constructing Footprints to query external data sources like Gene Bank and Swiss Prot. The Detente Inspector will provide access to these Footprints directly or may be used by other Inspectors to perform these external queries.

The Inspector provides a GUI to construct "query" matrices (associate genes, protein matches, etc.). Once a query has been built it can be targeted at one or more data resources and will be appropriately converted for each source. The Inspector then starts the queries and waits for the return data. The returned data can then be directed into one or more Inspectors for automatic correlation or analysis. Once analysis is complete the results may be used for additional queries, then sent to the user requesting the queries via email, a data base insertion, or, if the user is waiting, directly starting the appropriate Inspector with the new data.

Although this Inspector has been constructed the complete Footprints for external data source querying have yet to be completed. The CWP intends to construct these Footprints as the needs of collaborating labs dictate. As the Footprints are developed this Inspector is expected to be considerably enhanced to allow very fine control over data mining.

5.6 IONgraph:

The IONgraph Inspector is built from a number of independent modules that perform spectrum display, manipulation, and sequence comparison to spectral data. This Inspector allows for a dynamic manipulation of mass spectrum data.

The Inspector is capable of dealing with input spectral data which is then displayed within a GUI. The GUI allows for manipulations of the spectral data such as (to list a few): background filtration, normalization to X scale, precursor extraction, top N peak extraction, peak smoothing functions, peak searches and peptide ion calculations. Additionally, the Inspector accepts a sequence that the spectrum may be representing and calculates the theoretical spectrum of the sequence and shows correlation data between the displayed spectrum and the theoretical spectrum (after it has been manipulated similarly to the displayed spectrum). The sequence calculations also allow the user to edit or input alternate sequences and will then perform the same calculations on user supplied sequence data.

This Inspector has greatly reduced the spectral analysis time for CWP and collaborating scientists. The scientists who use this application estimate it saves at least 100 hours of data assembly work per experimental analysis.

5.7 Sammengi:

The Sammengi Inspector extends early NIH funded research and development of the quantitative analysis and comparison of 2D gel images of proteins [Garrels, 1988; Garrels and Franza, 1988a & b]. This Inspector is being designed to emulate and substantially expand the data representation and analytical functions provided by 2D gel analysis packages like QuestII and Galtool.

The Sammengi Inspector will accept images of 2D gels, or other shaped densities such as the “bands” in a 1D protein gel image or DNA sequence image, or “spots” from a DNA array image. These images can then be selectively analyzed via one or more automated methods in full three dimensions (length, breadth and depth of grayness, or OD). The Sammengi Inspector will import and analyzes any images captured from a wide variety of imaging devices provided a Footprint is available for that device output data type. Encoding such Footprint class extensions is straightforward and described above. The Inspector allows the user to fine tune the analysis through GUI based editing and input adjustment features. The peaks may be adjusted manually or automatically for use on awkward gels with spots of unusual profiles (e.g. the spots of some gels accept silver stain in an uneven manner). In addition to spot detection virtually all types of lanes can be analyzed including horizontal, vertical, overlapping and variable sizes.

The Sammengi Inspector is currently on hold due to resource constraints.

5.8 Visualize Mass Spectrum:

The Visualize Mass Spectrum Inspector supports analysis of protein identification derived from mass spectrometric analysis of peptide fragments. This Inspector also provides for analysis of these specialized applications reports in a single comprehensive application.

We collaborated with John Yates and his group to build an interface to the output data resultant from the SEQUEST analysis tool. The Visualize Mass Spectrum user interface supports dynamic sorting of the output data and other editing and display features. It is currently being tested by members of the NIH Resource Center for Comprehensive Genetics and was made generally available in the third quarter of 1997.

5.9 MolUnit:

In collaboration with Isis Pharmaceuticals CWP built a set of Imprint & Footprint classes which allow the import and export of molfiles. Molfiles store information describing the structural relationships and properties of a collection of atoms. Such collections may be referred to molecules, molecular fragments, substructures, substituent groups, polymers, etc. The resulting Imprints provide an object representation of those collections with ability to

visualize their 2D chemical structures, to build dynamically new chemical structures on a computer screen from a set of structures in separate panels and to store new structures as molfiles.

5.10 OligoChemistryImporter

CWP in collaboration with Isis Pharmaceuticals built a set of Imprints to represent oligo chemistry nucleotide, and an Inspector to read in 1 to N oligos from a set of specifically designed files for Isis' needs. The OligoChemistryImporter Inspector then runs extensive computations to confirm that each oligo is chemically valid, built from available subunits (as inventoried in a large Oracle database repository), computes a number of chemical cross references and finally stores the validated and cross referenced oligos in the Oracle database repository. This Inspector runs in parallel and on a database upgrade (moving all of Isis' oligos from an older database technology to the new Oracle database) processed 90,000 oligos in two days. Additionally it found over 1000 oligos which were stored in the older database that were actually invalid. This Inspector is used daily to validate 100 to 500 oligos in Isis' ongoing operation.

6.0 Example Use:

Several of the scientists participating in and with the NIH Resource Center are performing mass spectroscopic analysis of proteins. The description of each experiment and the resulting data and data analysis must be centrally archived. As mass spectrograms are generated and the analysis is performed, the resultant data files are centrally stored. The scientist performing the experiment is then able to access all the pertinent files from whatever computing device they use [PC, Mac, Unix host] without concerning themselves with the installation of specialized software, maintenance of the data files, or backup of both primary and analyzed data. They simply address the relevant experiment and its associated data through a "Web Server" interface that the Information Management group within CPB provided to them. From that page they have secure access to the index of their experiments, and by "pointing and clicking" they can traverse from index to data files, and, when required, launch the Moulage Inspector[s] required to visualize and further analyze their data.

When they complete a data viewing and analysis session, they can save and store what they have completed so as to access it again and/or to provide suitable pointers to members of their group or collaborators enabling them to access the files. The "public messaging" feature described above will provide a convenient way for group members and collaborators to share data and analysis without having to store and ship bulky files and documents on their host computers. The latter feature substantially reduces the individual scientist's cost of storing information and has the added benefit of secure, routine backup of all files.

7.0 Conclusion:

The Moulage System is a application framework designed to provide a scalable approach to data acquisition, analysis and mining while ensuring data integrity. By using object oriented design principles and the Moulage System foundation, a number of advanced applications have been constructed and proven extremely effective in about half of the development time normally associated with such projects. In fact some of the applications built, which consequently have shown to be tremendous work savers, would not have been considered for construction if the Moulage System had not provided the majority of programming work. While the Moulage System is still evolving and improving it has already proven to be a useful development tool.

8.0 Awards and Honors:

In 1997 the Moulage System and its scientific applications were used by Oracle as part of their 1997 Oracle Developers Conference for the keynote address.

In 1998 the Moulage System was a "1998 Computerworld Smithsonian Award" laureate in the "Education & Academia" category.

9.0 Copyrights:

The Cellworks Project work and web site was supported by an NIH/NCRR Resource Center Grant and by an NSF Science and Technology Center Grant.

Development of OligoChemistryImporter is supported by a contract from Isis Pharmaceuticals, Carlsbad, Ca.

All contents of this paper are Copyright University of Washington, 1995, 1996, 1997, 1998, 1999. All rights reserved.

CellWorks, Moulage System, Inspector, Caster, Footprint, Imprint and ImprintSorter are all trademarks of the University of Washington, Department of Molecular Biotechnology.

Java and HotJava are trademarks of Sun Microsystems, Inc., and refer to Sun's Java programming language and HotJava browser technologies.

10.0 Licensing:

The Moulage System source and binary executables are freely available for all academic and education institutions for any use. Businesses and individuals may freely use the Moulage System source and binary executables for development and testing purposes only. For nonexclusive commercial license to use or redistribute the Moulage System please contact the University of Washington Office of Technology Transfer (206-543-3970) or the University of Washington School of Medicine Office of Industrial Relations (206-685-8710).

11.0 Contributors:

- 1) Eduardo Firpo Ph.D., University of Washington
- 2) Alan Goates, Isis Pharmaceuticals
- 3) Raymond Kong, University of Washington
- 4) John McNeil, Isis Pharmaceuticals
- 5) QingHong Zhou MD, University of Washington