The Moulage System

(or its easier when its already done)

"***It's noble to be good. It's nobler to teach others to be good, and less trouble.***"
- Wit and Wisdom of Mark Twain, Alex Ayres

By Scottie Swenson
Cellworks Project
University of Washington

Notes:

Moulage System

- **Moulage:**
  1) A mold, as of a footprint, for use in a criminal investigation.
  2) The making of a moulage
  3) **a real time data sharing & analysis platform**

Notes:

The Moulage System is a real time data sharing & analysis system framework. Applications built using the Moulage System 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. Moulage System based applications can also provide access to high performance data systems to allow large computational analysis of data at speeds normally inaccessible to the Principal Investigators (PIs).

Approach

A. 100% Core Java (1.1)

a. Any needed classes built on the 1.1 core

b. Carefully engineered to work within the constraints

B. Provide All Foundation Functions

a. IO (File, Network, Database, etc.)

b. Data type identification

c. Resource controls (CPU, threads, Memory)

d. Data Management (real time databases)

e. More ...

Notes:

The Moulage System is fully multi-threaded for parallel or multi-process operations. The design allows for the ability to perform multiple analyses, data loading, and data mining operations simultaneously. In order to perform multiple tasks simultaneously, the entire system must be capable of preventing single resource corruption by multiple processes attempting to update that resource at the same instant. The design enables the Moulage System to take full advantage of modern multi-process platforms and parallel operating systems.

Design Goals

- Transparent access
- Accept multiple data sources
- Allow easy sharing of data
- Keep data in form usable by object-oriented databases
- Provide multiple tools for analysis
- Dynamic version distribution

Notes:

a) Transparent access: Allow users to access the extensive data and analysis methods, dynamically, from local or remote host computers.

b) Accept multiple data sources: Allow importing and exporting from any source data, no matter what format it is in.

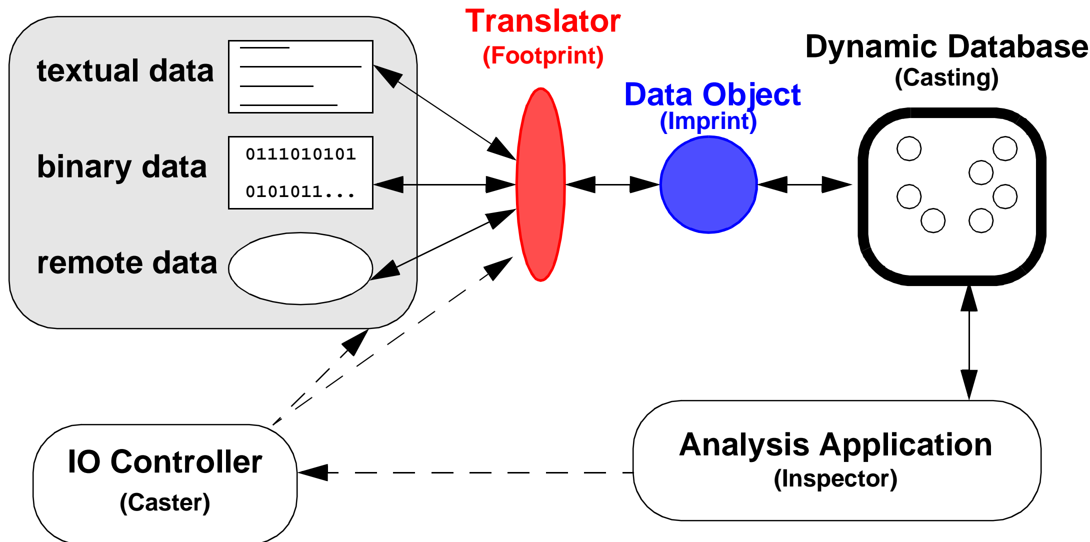
c) Allow easy sharing of data: Provide an environment that includes email, newsgroups, and other communication technologies, as well as data mining to either individuals or collaborating groups.

d) Keep data in form usable by object-oriented databases: Due to the nature of most items being analyzed, an object based approach is needed to allow for the diverse types of data. Since the data 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.

e) Provide multiple tools for analysis: Various data types can be analyzed with different applications, thereby requiring the system to ensure that the applications are compatible.

f) 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.

Object Interactions



Notes:

Moulage System components can be enhanced or replaced at the Moulage class level without affecting current systems. By carefully defining the application programming interface (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 sub-classes 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. The base objects are:

- Inspector- An application base. It is the central location for the user interfaces (GUI, command line, applet). Plus the location of the working logic needed to perform some specific analysis or function.
- Imprint - A data object type. Can be a simple atomic unit, or made up of other classes (including other imprints). It has some debugging and error handling methods built in and not much else.
- Caster - Black box utility for IO functions of the other classes. This class contains the ability to locate data in files, URLs, or in databases. It matches data to translators and vice versa.
- Casting - a database class for handling the data being manipulated by Inspectors. Designed as a drop in replacement for the Java.util.Vector class with lots of enhancements such as db querying abilities.
- Footprint - A translator class. It does not need to handle the IO as the Caster does this ahead of time. It does need to handle translating ONE file into one or more Imprints. It is also used in reverse to translate one or more Imprints back into a file.
- Utility - A class for any function that can be black boxed bean style.
- Spirit - A communications class that holds all of the environmental data for a given group of Moulage classes.

Uses at Cellworks

Started to be similar applications based on our internal classes linked to database server.

#1 - Visualize Mass Spectrum (VMS):

- Allow Scientists to review mass spec data without sitting in front of the instrument.

#2 - IONGraph:

- Enhance VMS to display actual spectral data.
- **Later**: Perform data gathering
- **Later**: Perform unique filtration on data
- **Later**: Allow multiple sets to be co-reviewed

Notes:

The Moulage System originally started out to be our internal mechanism for scientific applications. The applications would be used via a browser and linked to a central server running Oracle. The idea was to be able to send the applications with the data links back to our site. But, we knew the data manipulation had to occur on both ends (server and client). So the design was set up to be sectional to allow for the separation of functions between machines. The first application was visualize mass spectrum

The generic nature of Java took control. With each abstraction the classes began to look more like independent applications.

Uses at Isis

OligoChemImporter: Import oligo chemistry descriptions into central database

- Use the self-checking mechanism to validate chemistry
- Use deep self-checking mechanism to validate all aspects of chemistry
- **Later**: Added GUI to allow viewing the data as it loaded and process orders.
- **Later**: Use command line to integrate to a larger GUI application which overlays application

Notes:

At Isis the requirements where very different. They initially did not care about the GUI abilities of the application. What they needed was a command line application to suck in text files and process the chemistry definitions. Only later did they decide to use some of the GUI aspects.

Real World Benefits

Today:
- Unified API between applications
- Reduced development time
- Redundant functions already integrated
- Parallel processing architecture
- Automatic data validation

In "two weeks":
- Revolutionary integrated database
- Cross platform parallelism
- Advanced multi-tiered queue system integrated

Notes:

We are now integrating a new data management techniques along with advanced intelligent agents. The applications, applets no longer require the backend servers. Have not lost their applet and command line ability. But, have become something far more useful than the original plan.

Inspector

Application Foundation

Provides:

- Specific start-up method path
- Support for control applet parameter lookups (debug, Filebase, input files, filter index)
- Internal debugging system (debug output level controls)
- GUI for open file
- Pop Frame release from applets
- GUI for setting debugging output levels

Notes:

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). The Inspector class furnishes generic option controls and interfaces needed to perform data analysis.

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 called through a web browser 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".

Imprint

Data Object foundation

Built in:

- Error logging
- **Surface validation** (perstringe())
- ***Deep validation*** (battue())
- Automatic GUI object browsing

Notes:

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 CBP's OODBMS system.

The battue() method is a unique aspect of the CWP work. Our intention is to have a data system capable of redoing analysis and reconfirming the stored data on a regular basis.

Caster

IO Machine

Requests are sent in >> Databases are built
Nothing more

However it does:

- Queue requests fully multi-threaded
- Handle files, directories, URLs (in parallel)
- Regular expression parsing on all of the above
- Dynamic class loading of need filters
- CPU / System load balancing

Notes:

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 object 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 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 CBP 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.

Index Files - Version 1.0

Location,Index,Type KEY CLASSNAME

- First token is the location
- Second token is the KEY
- Last token is a footprint CLASSNAME

Notes:

A.The first token is the location specifier (l,i,f). The first two variables have different purposes depending on the value of the third token. The third token and the associated variables are interpreted in this manor:

1.Third token=t, then L=integer value indicating which line of the file to search to (counting from 1), and I=integer value indicating how many white spaced separated words into the line should be read to the KEY word.

2.Third token=b, then L=integer value indicating the offset into the file to seek() to (zero based), and I=integer value indicating the number of 8 bit bytes to read in from the offset position.

3.Third token=f, then L=binary operator of 1 for key match to end of the filename or 0 to match the key to the beginning of the filename, and I=binary operator of 1 to ignore the L variable and performs a regular expression a pattern match to the filename or 0 to perform a strict match governed by L (location).

B.The next token is the KEY or what to match at the location specified.

C.The last token is a footprint CLASSNAME which is at least the full package name of the footprint class to load for translation.

Index Files - Version 2.0

TYPE KEY[,KEY,...] CLASSNAME [URL]

- First token is the type of key (TEXT, BIN, FILE, NETCDF)
- Second token is the location and key
- Third token is a footprint CLASSNAME
- Fourth optional token is the URL locator

Notes:

A.The first token is the type of file specifier, which may be 'TEXT', 'BIN', 'FILE' or 'NETCDF'.

B.The second token is the KEY(s). In version 2 the KEY contains its location data AND optionally can be more than one key. The KEY token is a comma separated list of values that is based on the type of file being specified.

1.TEXT- For a 'text' file the KEY is: line#,word#,CaseSensitive,KEY. Where line# is an integer value indicating which line of the file to search to (counting from 1), word# is an integer value indicating how many white spaced separated words into the line should be read to the KEY word, CaseSensitive is a boolean represented by y/n/t/f (yes, no, true, false) which controls if the match to the KEY should be case sensitive. KEY is the string representing what the word is expected to be found.

2.BIN- For a 'bin' (binary) file the KEY is: offset,len,KEY. Where offset is the offset into the file to seek() to (zero based), len is the number of 8 bit bytes to read in from the offset position, and KEY is the uuencoded string representing what is expected to be found.

3.FILE- For a 'file' (filename match) the KEY is: matchType,KEY. Where matchType is the type of pattern being given options are: regexp (regular expression), begin (exact match to beginning of filename), or end (exact match to end of filename). KEY is the case insensitive string representing what is expected to be matched against the filename.

4.NETCDF- For a 'netcdf' file the KEY is: cdfType,name,KEY. Where cdfType is either "GA" (Global Attribute) or "V" (Variable), (NOTE: GAs work a lot better as keys.) name is the precise name of the GA or V, and KEY is the String representation of what is expected to be in the named variable or global attribute AFTER all white space areas have been converted to under bars ('_'). (e.g. the global attribute ':title = "Example Data"' would be represented by a key entry like netcdf GA,title,Example_Data footprints.netCDFDemo).

Both version 1 & 2 types expect the CLASSNAME to be specified as either the full package name of a class or as a URL locator with the full package name of the class.

Index Files - Version 3.0

TYPE,KEY[,KEY,...]<,&,|,!,>TYPE,KEY[,KEY,...]
CLASSNAME [URL]

- First token is the key list each key has a type key (TEXT, BIN, FILE, NETCDF) followed by the location and key. Additional keys are added with & (AND), | (OR) with the ! (NOT) modifier.
- Second token is a footprint CLASSNAME
- Third optional token is the URL locator

Notes:

Basically the same as version 2 with the addition of multiple keys being made available for matching a file type and the translator.

Casting

OODBMS (Mostly)

java.util.Vector (sort of)

A Casting:

- is a drop in replacement for java.util.Vector
- can only hold Imprints that pass perstringe()
- will only hold ONE Imprint type at a time
- will have a persistent storage mechanism that: significantly reduces storage size & search times; strongly validates that the data read is what was written; will be binary (compressed) or plain TEXT!!

Notes:

A Casting is a 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 a 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.

Footprint

Translator Foundation

As a java class the Footprint is:
- single minded: one type of file ONLY
- small: NO IO
- easy: 3 Methods and done

The hardest part is getting the file format right! (This is **NOT TRIVIAL**.)

Notes:

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:
- a short initialization method to describe the Footprint to the executing Moulage components;
- an import method which only needs to translate from an already loaded file to an already type checked Casting instance; and
- 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.

Utility

Workhorse Foundation

A small utility (bean) foundation meant to be developed and then used without further thought.

Basic premise:

- NO UI
- Defined ins and outs

Has the Inspector's: **debugging controls, event and execution methods**.

Has the Imprint's: **error logging**

Notes:

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.

Spirit

The Ghost in the Machine

Actually a special implementation of an Imprint.

Provides:

- Solution to applet context loss
- Analysis application grouping control
- Message handling between Moulage classes
- JVM shutdown

Notes:

There were a number of problems that demanded some hidden class. This class is NOT a singleton. There is only one per analysis group though. It contains all the settings, applet context data, control mechanisms for CPU/System loading, and event multicast handling for Inspectors.

Ensemble

Central Method Repository

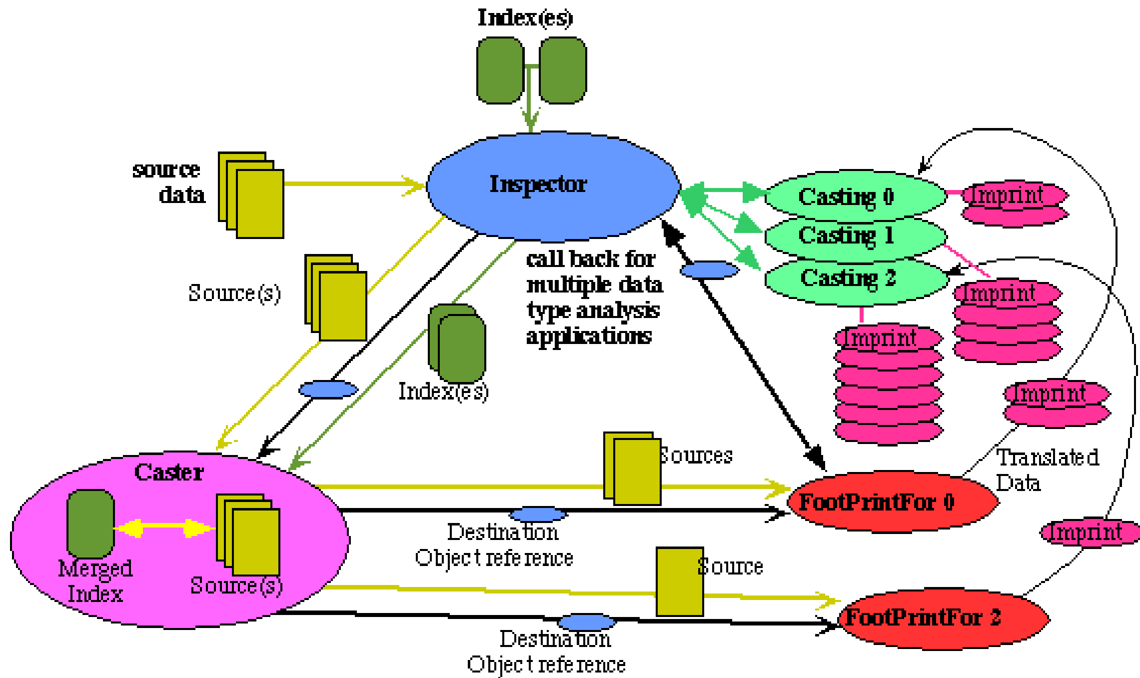
- Semi-OK fix for single inheritance limitations.
- Contains all of the Moulage group methods.
- All methods are static and thread safe.

Notes:

The only problem with Java's single inheritance is that you can not provide a clean method of multiple classes all having identical functions with some overrides. So this class has all of what would be in multiple inheritance and the various Moulage classes forward the function calls into this class.

No it is not perfect. But nothing else is either.

All Together - System Operation

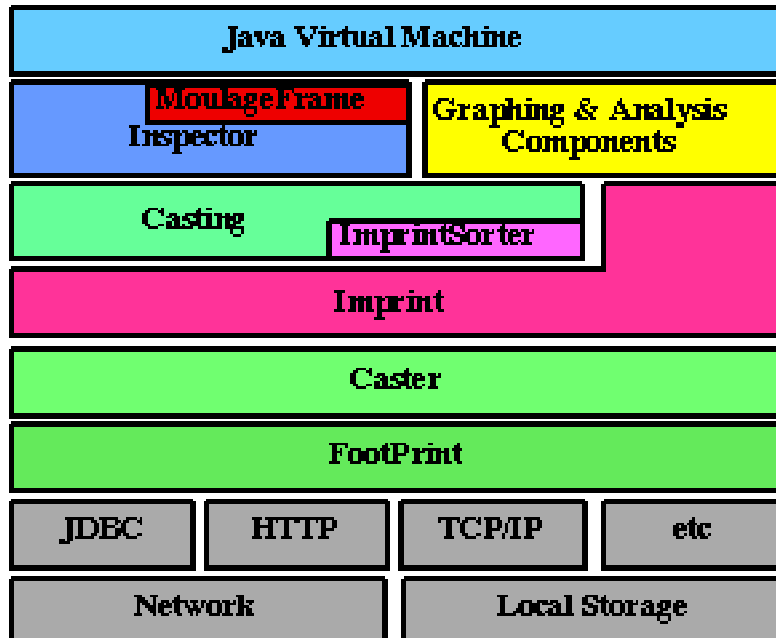


Notes:

All entrances to a Moulage System analysis are through an Inspector (the analysis interface for both real-time GUI interaction and automated analysis routines.) The Inspector is passed none, one or more optional inputs. The initial input is normally a list of source files, a list of index files to use for the data load, and debugging status. If the Inspector is running as a real time user interface, these items are dynamically modifiable by the user during execution.

Each data source is passed through the Caster, which identifies the source 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.

System "Layers"



- **Inspector**: data analysis
- **Imprint**: data object
- **Caster**: IO control
- **Casting**: database
- **Footprint**: data translator
- **Utility**: black box
- **Spirit**: Environmental holding bin

Notes:

Each of the base Moulage classes serve a very specific purpose. The basic structure of all of the CWP applications has been abstracted into specific elements. We'll go over each base class in detail. However, a short introduction to the whole picture will aid in understanding the various pieces.

- Inspector- the base for a specific application. It is the central location for the user interfaces (GUI, command line, applet). As well as the location of the working logic needed to perform some specific analysis or function.
- Imprint - light weight base for a specific type of data. It can be a simple atomic unit, or it can be made up of other classes (including other imprints). It has some debugging and error handling methods built in and not much else.
- Caster - a black box utility where all of the IO functions for the other classes resides. This class contains the ability to locate data in files, URLs, or in databases. It matches data to translators and vice versa.
- Casting - a database class for handling the data being manipulated by Inspectors. Designed as a drop in replacement for the Java.util.Vector class with lots of enhancements such as db querying abilities.
- Footprint - the base for a data translator class. It does not need to handle the IO as the Caster does this ahead of time. It does need to handle translating ONE file into one or more Imprints. It is also used in reverse to translate one or more Imprints back into a file.
- Utility - the base class for any function that can be black boxed bean style.
- Spirit - a special communications class that holds all of the environmental data for a given group of Moulage classes.