

Common Application Framework for Engineering Analysis (CAFEAN)

Preprocessor Plug-in API

Revision 1

Ken Jones
John Rothe
William Dunsford

August 2005

Table of Contents

1. Introduction.....	1
2. Preprocessor Plug-in Implementation.....	3
2.1 Plug-in Interface Classes.....	4
2.2 Plugin Interface Operations.....	6
2.2.1 Processing Batch Commands.....	6
2.2.2 Adding Menu Items.....	6
2.2.3 Submitting Jobs.....	6
2.2.4 Plugin Preferences.....	6
2.2.5 Essential Core Classes for a Code Plugin.....	7
2.3 The Multi-View Architecture.....	10
2.4 Creating a Model.....	12
2.4.1 Component Categories.....	12
2.4.2 Foreign Key Relationships.....	12
2.4.3 Methods to Implement.....	13
2.4.4 Model Options.....	15
2.4.5 Root Components.....	16
2.4.6 Component Number Groups.....	16
2.5 Creating Bean Based Components.....	17
2.5.1 Methods to Implement.....	17
2.5.2 The ComponentListener Interface.....	20
2.6 Creating Connections.....	22
2.6.1 Methods to Implement.....	22
2.6.2 Connection Drawing.....	23
2.7 ModelEditor Documents.....	24
2.7.1 The PIB Format.....	24
2.7.2 Loading a Model.....	25
2.7.3 Saving a Model.....	25
2.7.4 PibFile Load/Save Example.....	26
2.7.5 Model Load/Save Example.....	28
2.8 Undo and Redo.....	30
2.9 Creating Connectible Components.....	31
2.9.1 Methods to Implement.....	32
2.10 Plugin-Specific Unit Types.....	34
2.10.1 Supporting Units in the Model.....	34
2.10.2 Units Classes.....	35
2.11 Model Validation.....	37
2.11.1 ValidationTest Implementation.....	37
2.11.2 ValidationTest Methods in the Model.....	38
2.12 Using the Property View.....	39
2.12.1 The PropertyController Interface.....	39
2.12.2 Attribute Groups.....	40
2.13 Using Registered Dialogs.....	41
2.14 Customizing the 2D View.....	42
2.14.1 Adding Toolbars.....	42
2.14.2 Insertion Handlers and the Insertable Interface.....	42
2.14.3 Creating Custom Mouse Handlers.....	43

Table of Contents

2.15 Useful Utility Classes.....	44
2.15.1 Interfaces.....	44
2.15.2 Bean Editors.....	45
2.15.3 GUI Utilities.....	46
3. Packaging a Plug-in.....	47
4. Preprocessor Python Scripting.....	48
4.1 Built-in Python Methods.....	49
4.2 Core CAFEAN Classes.....	50
4.2.1 Real Class.....	50
4.2.2 AbstractComponent Abstract Class.....	50
4.3 TRACE Plug-in Examples.....	52
4.3.1 Hydraulic Components.....	52
4.3.2 Included files.....	55

1. Introduction

Applied Programming Technology, Inc. (APT) has developed the Symbolic Nuclear Analysis Package (SNAP) software under funding from the United States Nuclear Regulatory Commission (USNRC). SNAP is built on a highly flexible framework for creating and editing input for engineering analysis codes as well as extensive functionality for submitting, monitoring, and interacting with the analysis codes. This framework known as the Common Application Framework for Engineering Analysis (CAFEAN) provides a standardized application programming interface (API) used to create modular plug-in's for engineering analysis codes. The modular plug-in design of the software allows functionality to be tailored to the specific requirements of each analysis code.

This framework is essential to SNAP's Multi-View capability. The multi-view design permits several views of a component's data to be displayed simultaneously. These views can include 2D & 3D representations, editors, or displays of the analysis code ASCII input. Each of these views update automatically as the component data is modified. 2D views may also be embedded within other 2D views to provide a "Drill-Down" capability. CAFEAN also provides a framework to manage the complex interconnections that may exist between a model's components. This is accomplished by using a primary-foreign key relational mapping to manage component interconnections. By avoiding direct references between components, this approach provides the robust architectural base needed to support several of CAFEAN's more advanced features such as multi-step undo/redo, component duplication and cut&paste operations between models.

This document is intended to provide instructions for programmers who wish to extend the capabilities of SNAP by creating new plug-ins to support additional analysis codes and/or add new features to the user interface. Documentation of CAFEAN's python scripting capability which is useful for automating operations performed on models is also provided. Section 2 provides a detailed description of the implementation requirements for a Plug-in. Section 3 documents the packaging of a plug-in. Section 4 documents the scripting support available to the preprocessor client via a Python interpreter.

Who This Documentation is For

This API programmers guide is designed for someone with a working knowledge of Java™, Javabeans™, and the CAFEAN Preprocessor. The intention is for developers to be able to implement new analysis codes or feature operations for the Preprocessor without requiring modification of the primary source tree.

Why Make a Plug-in

The CAFEAN Preprocessor plug-in architecture provides a core functionality for the visual editing and manipulation of analysis code input files. All analysis code specific functionality is implemented using plug-ins. This approach allows support for new analysis codes to be developed independently of each other, allowing developers to

immediately implement necessary functionality without waiting for the primary source to be updated.

What Kind of Plug-in

The first task when developing a CAFEAN plug-in is to determine which type of plug-in to develop. The CAFEAN preprocessor plug-in architecture supports two basic types of plug-ins: code plug-ins and feature plug-ins. In general, if the new functionality includes new components, or provides support for a new analysis code, then it should be a code plug-in. If the plug-in manipulates existing components or models it is a feature plug-in.

2. Preprocessor Plug-in Implementation

This section provides an overview of the implementation requirements for a Preprocessor plug-in. The com.cafean.CodePlugins package contains the interfaces and abstract classes needed to develop plug-ins for the CAFEAN pre-processor, runtime and post-processor. Only those portions related to Preprocessor plug-ins are discussed here.



Figure 1. Plug-in Architecture UML

2.1 Plug-in Interface Classes

Preprocessor plug-ins are created by extending the abstract classes: MEPluginData, MEPlugin, MECodePlugin and MEFeaturePlugin contained in the nrcsnap.CodePlugins package.

MEPluginData Abstract Class

MEPluginData is the first class loaded by the plug-in manager is used to load the MEPlugin itself. MEPluginData contains methods for retrieving static information about all of the important values that define a plug-in. These values are: plugin-id, plug-in class, version, plug-in prerequisites, and class prerequisites.

- The plugin-id is the short name used to reference this plug-in.
- Plugin class is a String containing the absolute path of the plug-in class name including package names.
- Version is a String representing the current version of the plug-in.
- Plugin prerequisites is an array of strings that indicates which plug-ins need to be loaded for this plug-in to successfully load. Each of these strings contains the plugin-id and version of the plug-in it depends on, separated by a colon.
- The class prerequisites is the list of Java™ class packages that need to be installed on the users system for the plug-in to work properly.

MEPlugin Abstract Class

MEPlugin has several abstract methods that must be defined in any extending class. Some of this functionality has been implemented by MEFeaturePlugin and MECodePlugin, others must be defined by the plug-in.

There is some correlation between methods in MEPlugin and MEPluginData. Several methods are simply wrapper functions for methods in MEPluginData. This allows access to the requirements and information about a given plug-in after the plug-in has been loaded.

- getPluginId() should return the the plug-in data's plugin-id.
- getVersion() should return the version.
- getPluginPrereqs() should return the plug-in prerequisites in the form [Plug-in ID]:<Version>. The version (and accompanying colon) are both optional.
- getPluginInfo() should return a description of the plug-in. Plug-in info is used inside the AboutPluginsDialog for describing plug-ins to the user. It should be either unformatted text, or conservatively formatted HTML suitable for inserting into an existing HTML document.

MECodePlugin Abstract Class

MECodePlugins are the implementation of support for an analysis code in the ModelEditor. This implies a new AbstractModel extension, with its own set of components.

When the user wishes to create a new model in the ModelEditor, they are prompted to choose the type of model (by plugin-id) they want to create. createNewModel is called on the chosen plug-in and the model is added to the list of currently open models in the Mainframe and Navigator.

It is recommended that plug-ins store their models in ModelEditor Document (MED) files in a platform independent binary file format. For more information on loading and saving models refer to the ModelEditor Documents section.

MEFeaturePlugin Abstract Class

There are five methods that are needed by every MEFeaturePlugin: load, save, modelRemoved, modelAdded, and isAssociated. MEFeaturePlugins may store data inside the MED file produced by another model in the ModelEditor. This is done in such a way that the model can still be retrieved from the MED file even on a machine that doesn't have the MEFeaturePlugin.

Feature Plug-ins may have data related to a particular open model. When this relationship exists, isAssociated should return true for that model. When a model associated with a MEFeaturePlugin is saved to a local file, that plug-in is notified by the Mainframe through the save function. That save function is passed the model that is being stored, and the PibFile that is being written out. This allows the plug-in to write whatever blocks it needs into the MED file.

Similarly, when an MECodePlugin is reading in an MED file, at the end of its parse cycle, it may encounter information from a MEFeaturePlugin. When it does, the MEFeaturePlugin is informed through the load function. The load function is given the AbstractModel that has just been read in, and the PibFile. This allows the plug-in to load its model specific data back into the ModelEditor.

Since MEFeaturePlugins may use data that is specific to individual AbstractModels, these plug-ins are notified whenever an AbstractModel is added or removed from the ModelEditor. This allows the plug-in to dispose of any local data, or to create new local data as appropriate.

2.2 Plugin Interface Operations

Most of the user accessible functionality for a plug-in is implemented through toolbars on the MainFrame and DrawnViews and the batch command processor.

2.2.1 Processing Batch Commands

Commands are parsed by white space and converted to a vector of words prior to being sent to the batch command processor. Batch commands that contain a plug-in ID as the first word are forwarded to the respective plug-in's processCommand method. In this case, the plug-in ID element is removed from the vector prior to being sent to the plug-in.

2.2.2 Adding Menu Items

Menu items can be inserted by a plug-in into the Mainframe (in loadMainMenuItems) or the DrawnView (in loadViewMenuItems). For the DrawnView, addItem is used to add items to the Tools menu.

MainFrame's addItem allows JMenuItem's to be inserted into any of the main menus, which are selected by name. The name of the menu is the word that appears in the main toolbar for that menu: File, Edit, View, Window, Tools, and Help are the valid names. Menu items are inserted in the order the plug-ins are read in from the directory.

MainFrame's addImportMenuItem and MEPlugin's addCurrentExportItems allows the plug-ins to specify their import and export functionality. If there is more than one way to import a model, it is preferred for a JMenuItem to be added (with the plugin-id as the text), that contains all of the import operations for that plug-in. All menu items may make use of Mainframe's getCurrentModel to obtain the model that currently has editing focus. This allows the plug-in to determine whether the plug-in item should affect the current model, and respond accordingly.

DrawnView's addItem appends a menu item to the Tools menu that appears on every DrawnView. The DrawnView has a method for obtaining the model it represents, so it is easy to ensure that only certain types of models get plug-in menu items.

2.2.3 Submitting Jobs

When an AbstractModel is ready to be submitted to the Calculation Server, it is the MECodePlugin's responsibility to get it there. A detailed example of submitModel is provided in the class documentation of MECodePlugin. Additional information can be found in the documentation for LocalSubmitDialog.

2.2.4 Plugin Preferences

2.2.4.1 Loading and Saving Preferences

Each plug-in may store settings information into the program settings file. Before the plug-in gets an opportunity to load or store its settings, the Configurator switches to a module with the plugin-id as it's name. This prevents any possible overwriting of data in

the user settings for different plug-ins. The plug-in can then start directly loading and storing values in the Configurator. The settings file is written in XML format, so all values ultimately get stored and read as strings, but there are some convenience methods for storing specific types of data such as fonts and colors.

2.2.4.2 Editing Preferences

Plugin preferences can be made editable by building a JavaBean™ that contains properties for each of the editable preferences. This bean is then returned by the plug-in's `getPluginPreferences` method and edited in the Preferences Dialog in the same manner as other JavaBean™ type objects and components.

2.2.5 Essential Core Classes for a Code Plugin

AbstractModel Abstract Class

The `AbstractModel` is the heart of a `MECodePlugin`. The model contains all of the components, connections, and objects that are needed to model a simulation. All of the `AbstractComponents` inside a model are organized by `Category` inside of `ComponentLists`. Access to a component can be by its primary key (called the `ident`), component (`cc`) number, or by a secondary temporary key called the `db_id`.

The `AbstractModel` controls all of the model level operations. It starts save and load routines for the whole model, as well as initiates imports and exports.

AbstractComponent Abstract Class

Any object for an analysis code that can be displayed in a view or the Navigator should be an `AbstractComponent`. This number is usually unique within the components immediate category. Many analysis codes organize their components solely by component number. `ComponentListeners` are objects, or components that are notified whenever a change occurs with a given component or its connections.

GenericObject Abstract Class

Anything inside a model that is not a component, but needs a global primary key, is a `GenericObject`. A single `ElementList` and its accessors is provided by `AbstractModel` and can be used by subclasses to store any `GenericObjects` that may be needed for the model.

For example, cells contained within thermal-hydraulic components commonly extend the `GenericObject` class. This allows cells to be accessed using primary-foreign key relationships, eliminating the pitfalls associated with direct references and simplifying processes such as renodalization.

Connection Abstract Class

Connections are special `AbstractComponents` that connect two other `AbstractComponents` together. The generic form has only two primary key references,

and two Categories. Since Connection itself is an abstract class, appropriate extensions to the Connection must be implemented.

Connections are stored in a ComponentList in the AbstractModel. When a connection is completed, the Connection object created is added to both Components with their respective addConnection methods. Each AbstractComponent has a ConnectionList, which essentially is a list of foreign key references to the Connections stored in the AbstractModel. All Connections involving a component can be retrieved with its getConnections method and are normally stored in an instance of ConnectionList within that component.

ConnectionData Abstract Class

A ConnectionData object is transient data that is generated either by the DrawnComponent when it is building its ConnectingPt objects, or by a Connection when it is determining the nature of the connections on either side. Two ConnectionData objects can be used to initiate a connection between two components as well, using the AbstractComponent's connectTo method.

ConnectionData represents the actual location information on a component for a given connection. For example, for a HydroConnection between two pipes, the HydroConnectionData includes the cell index, the face number on the target component, the edge index, and face number on the source component.

Category Class

Categories provide the primary source of component organization within a model. They form a hierarchical representation in that a Category may be a subset or superset of other Categories. For example, the Category for all hydraulic components would be the superset for the Category for pipes as well as the Category for pumps. Each category contains the icon for display in the navigator, the URL of the image for the Toolbox, the name of the category, and whether the category represents visual components. Parent categories and non-visual components do not need a URL for their toolbox image, and non-visual components do not need an icon for the navigator.

AbstractModel contains predefined categories for views and connections. They are publicly defined in AbstractModel as CAT_VIEW and CAT_CONNECTION, respectively. These Categories and their lists are handled by AbstractModel and need not be handled by derivative models.

NOTE: Categories must be compared by isSubset, isSuperset or equals, not with reference equivalence.

ComponentList Class

The ComponentList is a list of AbstractComponents sorted by their primary key. ComponentList does not allow duplicate keys but will allow a single object to be added

multiple times. All searches on this component list by that key are done using a binary search algorithm.

DrawnView Class

The DrawnView is the dialog that contains the actual View. Within this dialog the ZoomablePanel maintains a zoomed view of its contained BeanBox. The BeanBox in turn, contains and displays the DrawnComponents.

Menu items can be added to a DrawnView's Tools menu from a plug-in's loadViewMenuItems method by using DrawnView's addItem method.

DrawnComponent Abstract Class

Most AbstractComponents can be rendered in a DrawnView. Each AbstractComponent has its own implementation of createDrawnComponent(). This returns a DrawnComponent capable of rendering the AbstractComponent that created it. The DrawnComponent is the rendering engine for the component in the two dimensional DrawnView. The DrawnComponent also contains ConnectingPts, which are locations on the drawn icon where a connection can be started or completed. The local information of what that ConnectingPt actually represents in the component (such as cell number or face value) is stored in that ConnectingPt's ConnectionData object.

If a ConnectingPt represents multiple internal locations that the user must choose between when a connection is completed, a SpecialConnectionData object should be used instead. When a connection is completed by the user, each component gets the opportunity to modify the ConnectionData object obtained from the ConnectingPt on it's DrawnComponent with the createTargetData and createSourceData methods.

When a connection is rendered using a DrawnConnection, it is drawn between two ConnectingPts. This is done by comparing all of the ConnectingPt's ConnectionData objects until a match is found using ConnectionData's equals method.

ConnectingPt Class

A ConnectingPt is a point on a DrawnComponent that can be the source or target of a Connection. They are created with DrawnComponent's createConnectionPt and used by the connection tool to find potential connection beginnings and endings as well as by DrawnConnections to find the location of their end points.

Each ConnectingPt has a ConnectionData object that describes the correlation between its location on the DrawnComponent and the Component it represents. It also has a Pad object that defines the ConnectingPt's position and the orientation of any lines exiting the DrawnComponent at this point.

2.3 The Multi-View Architecture

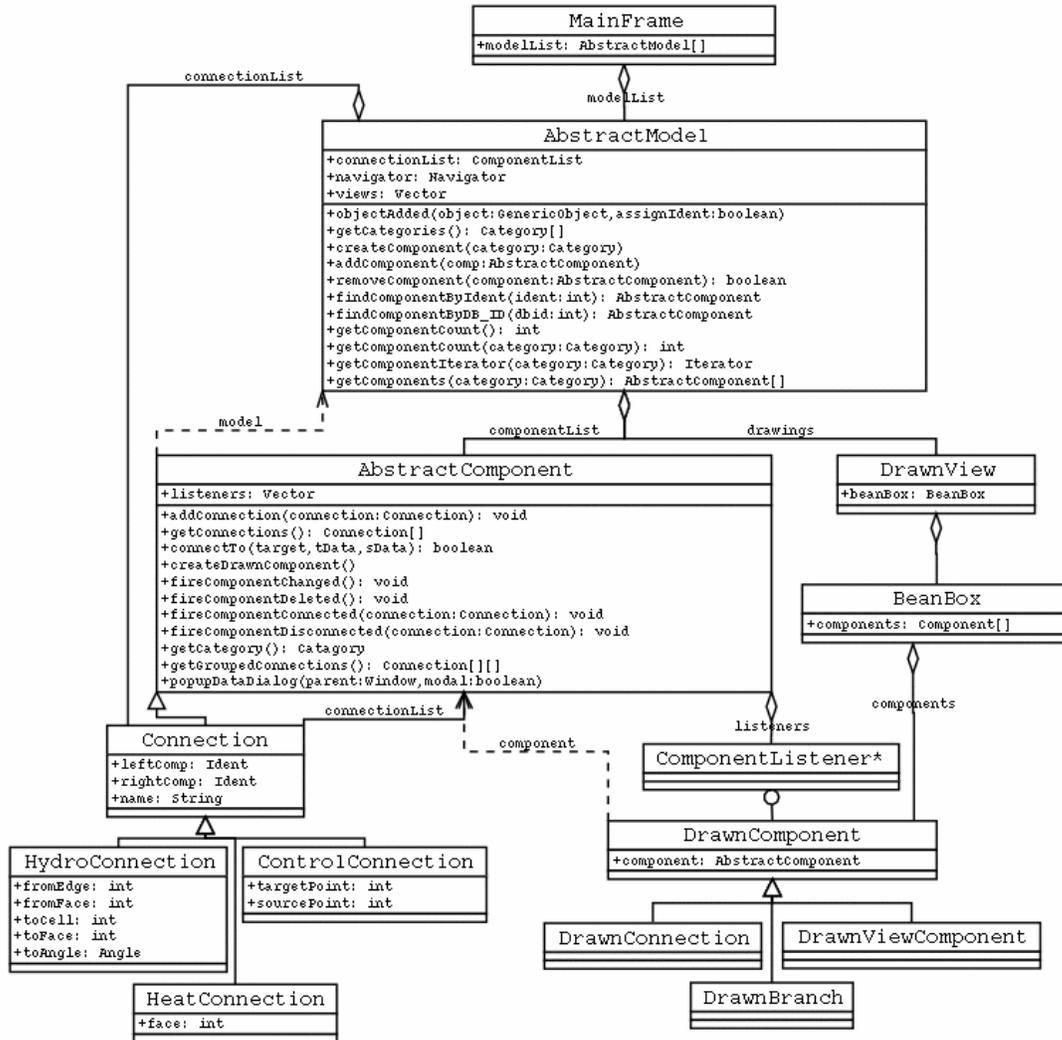


Figure 2. Multi-View Architecture UML

Figure 2 illustrates the ModelEditor's Multi-View architecture and component management features. In the ModelEditor all components are AbstractComponents, and all AbstractComponents are stored in ComponentLists in the model. These components are then referred to by a unique id called an ident.

Models also have a hierarchy of categories which each AbstractComponent falls into. For each component type there is a Category, and these component categories are grouped into other categories.

For instance: A TRACE pipe is in the *Pipes* category, which is in the *1D Components* category, which is part of the *Hydraulic Components* category. When other 1D and hydraulic components are taken into account a very organized hierarchy of categories

results. These Category objects are used for searching for, iterating through or retrieving all of particular groupings of components. They also determine the structure shown in the Navigator.

2.4 Creating a Model

The model is the central data structure for any Code plug-in. It handles the creation, loading and saving its contained components. As a result, a plugin's AbstractModel extension requires a significant amount of customization to be fully functional. The following is a discussion of the structures, concepts and methods required for a complete model.

2.4.1 Component Categories

Components in the CAFEAN preprocessor are organized into a hierarchical set of groups called *categories*. Each *category* is represented by an instance of the Category class. Each Category can have a parent and a set of child Categories. With this system, each component type should have its own Category instance. Take note, however, that a Category that represents an actual component cannot have child Categories.

The Categories made available by a model are used in various places in the UI to organize and select components. For example: The Navigator's tree structure is determined almost entirely by the Category instances returned by the model's getCategories() method. Also, a component's Category is used to create instances of that component from the Navigator via the model's createComponent(...) method.

Note that if user defined numerics or views are to be included in the Navigator's listing of components for a model, these categories (CAT_NUMERICS & CAT_VIEW) must be included in the array returned by getCategories().

Refer to the Source Code Documentation of the Category class for more information and detailed examples of the creation and use of Categories.

2.4.2 Foreign Key Relationships

Each component in the CAFEAN preprocessor contains a unique primary key called an *ident*. Foreign key references to these ids are used for all component-to-component references in the ModelEditor. For simple one-way references, a single foreign key reference is used. For two-way references or references that must be rendered as lines in a DrawnView, a Connection object is used that contains foreign key references to both components.

In all cases the ident reference value 0 indicates an invalid reference (i.e. NULL) thus no search is required to determine the component referred to. All non-zero ident reference values are considered valid and require a search to determine the component referred to. Note that a non-zero ident reference does not necessarily mean that the referred to component exists or is available.

Refer to the Source Code Documentation of the AbstractModel class for more information of the handling of foreign key references.

2.4.3 Methods to Implement

AbstractModel contains a set of abstract methods that must be overridden. It also contains a larger set of methods that may also need to be overridden for particular Code Plug-ins. The following is a list of the methods that must be overridden and their purpose as well as a list of optional methods that are commonly used in Code Plug-ins.

The following methods must be implemented by every model. Some are required before the resulting class will compile; others are implemented to handle the components stored in AbstractModel and must be extended to handle the new model's components.

GetPluginId Method

This method is a simple accessor for the the model's Plugin-ID. It should return the same value as the MEPluginData included with that plug-in.

saveModel Method

A version of this method is provided both with and without a showProgress parameter. This parameter indicated whether or not a progress dialog should be shown during the save process.

loadRestartData Method

This method is used to retrieve data from the CAFEAN Calculation Server in order to initialize the current model with a set of restart data. This data normally consists of initial conditions for submitting a restart.

checkModel Method

This method should perform a set of checks on the model and its' component's current state. This method normally utilizes AbstractComponent's isOkayForExport methods and may optionally add error and/or warning messages to the MessageWindow.

For more information on checkModel, refer to the Model Validation Section.

getCategories Method

This method retrieves the complete set of Categories available for this model and should be overridden to include any new Categories defined by the model. The ordering of these Categories is used by the Navigator to determine it's tree-node ordering.

Note: This set should include the categories defined by AbstractModel that will be used in this plug-in. (e.g. CAT_NUMERICS, CAT_VIEW)

getComponents Method

This method must be overridden to handle any new Categories. This method should be able to handle a parent Category by calling `getComponents` with each of its children. Note that Categories defined in `AbstractModel` should be passed to `AbstractModel`'s implementation of `getComponents`.

findComponentByIdent / findComponentByDB_ID Methods

These methods retrieve a component by the `ident` or `DB_ID` (and optionally `Category`) given. Only the version with a `Category` parameter should be overridden. These methods, like `getComponents`, must be overridden to handle any new Categories in this model.

For more detailed information on either of these methods, see the [Source Code Documentation](#).

addComponent Method

This method adds a component to the appropriate internal `ComponentList` and ensures that it has an appropriate `ident`. The default implementation of `addComponent` handles those Categories defined in `AbstractModel`.

This method must be overridden to store new component types into an appropriate `ComponentList` (defined in the model) and to handle updating the `ident` of the added component with a call to `objectAdded`.

Note: Plug-ins with a small number of component instances (hundreds) may choose to use a single `ComponentList` instance to hold all components.

createComponent Method

This simple method takes a `Category` and creates an appropriate component instance using that component's default constructor. This must be overridden to handle new Categories and should call the default implementation for those Categories defined in `AbstractModel`. The model may be set on the component but the component should not be added to the model.

removeComponent Method

This method removes a component from the `ComponentList` it is stored in and calls `fireComponentDeleted` on that component. This must be overridden to handle new Categories and should call the default implementation for those Categories defined in `AbstractModel`.

getComponentIterator Method

This creates an `Iterator` for use in traversing a subset of the components in the model. The default implementation uses the `ComponentList` method `iterator()` for the appropriate

ComponentList. The Iterator created will traverse the components that are part of the Category given in ident order. This must be overridden to handle new Categories and should call the default implementation for those Categories defined in AbstractModel.

Note: The current Iterator creation strategy cannot span multiple ComponentList instances.

getComponentCount Method

This method returns the number of components in the model who's Category is a subset of the given Category. This, much like removeComponent, must be overridden to handle new Categories.

reconnectIdentReferences Method

This method calls reconnectIdentReferences on every foreign key holder in the model. The default implementation handles only the ComponentList instances defined in AbstractModel. This must be overridden to handle any new ComponentList instances as well as the model's options object if it contains foreign keys.

For more information on reconnectIdentReferences, see the Source Code Documentation for AbstractModel.

clearDbIds Method

This method clears the DB_IDs of all components in the model. It must be overridden, like reconnectIdentReferences, to handle any new ComponentLists.

getModelOptions Method

This retrieves a plugin-specific model *options* object that is assumed to be a JavaBean™. If this method returns null then the options node will not appear. The default implementation returns null and does not necessarily need to be overridden. See Model Options below.

layoutComponents Method

This method is used to organize and layout DrawnComponent instances in a View. This is a plugin-specific method intended only to determine appropriate x,y locations in the View for each DrawingComponent given.

2.4.4 Model Options

A model's *options* object is intended as a place to store properties that are specific to a particular model but not complex or large enough to justify their own component. Properties such as a model's name, description comments should be stored here. This object is assumed to be a JavaBean™ and should extend AbstractBeanComponent to ensure that ComponentChanged events are generated when it is edited.

The Model Options node will appear as the first sub-node of the model in the Navigator. This node's string and icon representation will be taken directly from the BeanInfo of the options object. This is essentially the same functionality offered by using Root Component with much smaller implementation requirements.

2.4.5 Root Components

The root components feature is intended to handle cases where there is only a single instance of a particular component in a model. Root components are displayed in the Navigator just below Model Options (if present) using the same node type as other components. The component's toString method is used as the name of the node and it's Category decides its icon. Root components will appear in the Navigator in the same order as they are returned from getRootComponents.

Plugins requiring root components must override AbstractModel's getRootComponents method and handle reconnecting any foreign key references contained in those components.

2.4.6 Component Number Groups

Component number groups are used to allocate component numbers to new components, to determine if any existing numbers are invalid or duplicated and to renumber sets of components within a specified range.

To use component number groups in a code plugin, the GetComponentGroups method must be overridden to return appropriate groups for the available component types.

For each group the following must be defined:

- the initial component number
This is the first number available in the group. The first component created in this group will receive this number.
Note: This number is not necessarily smaller or lower than the maximum.
- the maximum component number
This is the last component number available in the group. This is not necessarily higher than the initial number.
- the increment quantity
This is the amount added to the last used number to determine the next allocated number. To create a group that begins at -1 and goes to -1000 set the initial to -1, maximum -1000 and increment to -1. (or any negative integer)
- the Categories included in the group
This is the set component Categories that will be included in any checks and renumbering performed with this group. These Categories will also be used to determine which group a particular component is part of when allocating a new component number. Note that a group may have one or more Categories.

2.5 Creating Bean Based Components

Components are the basic data structures for all existing code plug-ins. With this in mind, care must be taken when determining what analysis code structures map to components. In general, if there can be more than one of a particular object and/or that object is referred to by other components then that object should be a component. Often if there is only one of a given component it maps well to a root component. Root components and non-root components are both components and thus the following section applies equally to both.

To be a component in the CAFEAN pre-processor, an object must extend `AbstractComponent` or `AbstractBeanComponent`. Components in a `JavaBean™` based model should extend `AbstractBeanComponent` to enable access to the more modern bean based editors and undo available in CAFEAN. In addition, appropriate `BeanInfo` classes must be provided for each component to allow proper display of properties for that component in the Property View.

All new plug-in development should follow the `JavaBean™` architecture and use the bean based CAFEAN functionality. The creation of non-`JavaBean™` plug-ins is not recommended.

2.5.1 Methods to Implement

label Method (*Required*)

This simple method should return the name of this component's type. For example: "Pipe" or "Pump". The default `toString` method for `AbstractComponent` uses this method, the component number and the component name to create a simple string representation.

The default implementation of this method uses the class name as the label.

getCategory Method (*Required*)

This returns a reference to the `Category` this component is part of. The `Category` will be used for foreign key searches and adding this component to a model. This is the only abstract method in `AbstractComponent` and is essential. See the discussion of `Categories` in the model creation section as well as the `Source Code Documentation` for `Category` for more information on the use and creation of `Categories`.

clone Method (*Required*)

`GenericObject` (the ultimate parent of `AbstractComponent`) is a `Cloneable` class, therefore all components and their sub-components must have proper clone methods defined. Refer to the `Java` documentation for more information on how to properly implement clone.

storeState / restoreState Methods (*Required*)

The CAFEAN pre-processor undo system assumes that all objects being edited are JavaBeans™ and that all beans are StateEditable. Because of this, proper storeState and restoreState methods must be implemented for all components and sub-components. Improperly implemented storeState and restoreState methods can cause errors that are difficult to diagnose. Refer to the Java documentation for more information on how to properly implement storeState and restoreState.

complete Method (*Optional*)

This method is used to properly initialize components after they've been created either via the Navigator or the Insert Tool. A call to this method implies that the user has created a new component and that the created component has been added to the model.

Completion operations that do not require user input can be freely executed here. If completion requires GUI interaction (such as using an OptionPane), a SwingWorker must be used to avoid a deadlock on the Swing Event thread. Refer to the Source Code Documentation of com.cafean.utils.SwingWorker for more information on UI interaction using the SwingWorker.

isOkayForExport Method (*Optional*)

This method determines if the component has valid data for export. If the *prompt* parameter to this method is true, the method should add error or warning messages (via addMessage) for each of the problems found during the validation. This method should be called on each component from the model's checkModel method.

See the Source Code Documentation for MainFrame for more information on addMessage.

getCustomPopupMenu Method (*Optional*)

This method and the following getCustomPopupActions can be used to customize the popup menus that are created for a particular component. This should return a Vector of JMenuItem, JMenu, and JSeparators representing the complete pop-up menu for this component.

The default implementation includes:

1. The *Show ASCII* item for Writeable objects.

This creates an AsciiViewer to display the written representation of the component. See the Useful Utility Classes section for more information on the Writeable interface and it's use.

2. The *Reference Docs* menu.

See the Reference Document Links section for more information on how to define and use Reference Documentation.

3. The *Special* menu.

This menu is built by creating items and separators from the actions returned by `getCustomPopupActions`.

4. The *Properties* item.

This item creates a Mini-Navigator for the selected object (In this case a component) that can be used to edit the object's properties. This dialog acts in most ways like the Navigator and Property View.

Additional menus and menu items can be inserted into the vector by component extensions.

getCustomPopupActions Method (Optional)

This method returns an array of Action objects that are used by `getCustomPopupItems` to create its *Special* menu. The Special menu is intended to contain operations that can be performed on a component that are used rarely or in special situations. Placing less used operations here can greatly shorten and simplify the component's popup menu.

The Action array returned may contain null entries to indicate separators.

removeVerify Method (Optional)

This method is called by the UI to verify that the user may remove the component from its model without unforeseen side effects. For example: Removing a shared *geometry* object used by a dozen heat structures. Returning true from this method implies that the component removal should be allowed.

This method may request verification from the user via `OptionPanels` or dialogs.

getOrder/setOrder Methods (Optional)

These methods are accessors for a relative ordering value. This can be used to ensure a particular sorting order for a set of components. This is the primary attribute used in sorting when using the Comparator returned by `getOrderComparator`. This has primarily been used by plug-ins to preserve the relative ordering of components in imported decks by setting this value on import and using the order comparator to sort the components before exporting them.

toString Method (Optional)

This method returns a string representation of the component. The default implementation returns a string in the form: `<label> <component number> (<name>)`.

2.5.1.1 Useful Utility Methods

writeName Method

This method returns a string with its surrounding whitespace removed. If the resulting string is "unnamed" (the default name) then an empty string is returned instead. This was intended for use in an ASCII export of a component.

popupDataDialog Method

This method creates and shows an editing dialog for the component. For JavaBean™ based components a Mini-Navigator will be created with a Property View on the bottom. This Property View acts identically to the main Property View and the two can be used interchangeably. This method may be overridden to create a different editing dialog for a particular component.

2.5.2 The ComponentListener Interface

Various structures and views in the CAFEAN ModelEditor use the ComponentListener interface to appropriately respond to changes to components. The Property View in particular uses this interface to ensure that the properties displayed reflect what is stored in the component.

2.5.2.1 AbstractComponent Methods

The following AbstractComponent methods relate directly to the ComponentListener interface. Any or all may be overridden if necessary but the default implementation must be called from the overriding method.

addComponentListener Method

This adds a structure to be notified when this component is changed, deleted, connected or disconnected. The given listener will be compared to the current list to ensure that each is added only once.

removeComponentListener Method

This removes an existing listener from the list of structures to be notified. Attempting to remove a structure that is not currently a listener will not cause an error.

fireComponentChanged Method

This method notifies all structures listening to the component(ComponentListeners) that its data has changed in some way and should be reloaded or refreshed. This method is called by the Property View, the undo system and various other operations.

fireComponentDeleted Method

Much like `fireComponentChanged`, this method notifies all listeners that this component has been deleted. This indicates that views of this component should be closed or cleared, Navigator nodes removed, etc.

fireComponentConnected / fireComponentDisconnected Methods

Like the previous two methods, notifications are sent of connection and disconnection. Often the disconnection also causes a *deleted* event to be sent for the Connection. The connection affected should be passed to this method to allow listeners to update appropriately.

2.5.2.2 ComponentListener Methods

To use the `ComponentListener` interface to track changes in a component, the following methods must be implemented. Refer to the Source Code Documentation for the `ComponentListener` interface for more information.

componentChanged Method

This is a notification that the component described in the given event has changed. The event itself may describe the change in more detail. See the Source Code Documentation for `ComponentChangedEvent` for more information on available extensions. Plugin-specific extensions of this event may be created and used to describe other types of changes.

componentDeleted Method

This is a notification that this component has been deleted and views must update and/or close themselves. This is particularly useful for such things as ASCII views and editing dialogs for components.

componentConnected / componentDisconnected Methods

This is a notification that the Connection given has been connected or disconnected. This has been used to update data related directly to connections, to update `DrawnComponent` decorations and to update ASCII views.

2.6 Creating Connections

A connection is any object that extends `ConnectionBean` and is used to represent an association or connection between two other components. In essence it's an encapsulation of two ident references with an object describing each reference. Since connections are also components, much of the section on creating components also applies to connections.

An important decision when determining if an object should be mapped to a connection is whether that object can exist in a `View` without the components it connects to. Currently, connections will only be included in a view (if the components on either end are included). This enables connections to be represented as lines drawn between the `DrawnComponents` representing the components on each end.

2.6.1 Methods to Implement

The following methods are important to the implementation of a `Connection` object. Of the methods listed, only the first two must be implemented to create a functional `Connection`.

Refer to the Source Code Documentation for `Connection` and `ConnectionBean` for more detailed information on the the methods available for overriding.

getLeftConnectionData / getRightConnectionData Methods *(Required)*

These methods retrieve the data object for each side of the connection. These connection data objects describe the relationship of this connection to the component on each side. For some connections there may be no need of this descriptive data; the connection itself may be enough. In these cases an empty `ConnectionData` extension can be returned.

Refer to the Source Code Documentation for more detailed information on use an creation of `ConnectionData` objects.

IsIndependentComponent Method *(Optional)*

This method is used to determine whether the connection can exist without the components at either end. Non-Independent connections are disconnected and removed if and when their surrounding components are no longer available. If the connection type has enough data to justify its existence without being connected then it should return true here.

label Method *(Optional)*

`Connection` overrides `label` to return the string "Connection." If this is appropriate for the connection in question then this method does not need to be overridden.

getDocDescription Method *(Optional)*

This method is used primarily by the ConnectionSetPanel to build an HTML description of this connection, the components on either end and the connection data describing each side.

getCustomPopups Method *(Optional)*

This method is used the same way as it is in AbstractComponent derivatives. The default implementation in AbstractComponent has been overridden here to include only the Disconnect item.

userDisconnect Method *(Optional)*

This method is called by the UI to disconnect this connection. This may be overridden by connection types that require user interaction or additional processing to be completed when the user disconnects a connection instead of a programmatic disconnection. The default implementation simply calls disconnect.

2.6.2 Connection Drawing

By default, a visual connection will be represented in a View by a DrawnConnection object. DrawnConnection uses the following methods to determine the appearance of the line drawn between the connected components.

isVisual Method *(Optional)*

This method is used to determine if a DrawnConnection will be created for this connection if both sides of the connection are present in a View. The default implementation returns true.

getConnectionColor / getConnectionStroke Methods *(Optional)*

This is used by DrawnConnection to get the color and stroke to use when drawing this connection. By default, the color used is the "Connection Color" in global preferences and the stroke is a new instance of BasicStroke.

createDrawnComponent Method *(Optional)*

This method creates a new DrawnConnection configured for drawing this connection. Extensions may override this method to alter the appearance of the connection. See the section on creating DrawnComponents for more information on how to create and use DrawnComponents.

2.7 ModelEditor Documents

It is recommended that plug-ins store their models in a platform independent binary ModelEditor Document (MED) file format. The remainder of Section 2.7 details the implementation of loading and saving logic for a model using a PIB based MED format.

2.7.1 The PIB Format

Platform Independent Binary (PIB) files consist of a file header section followed by a series of named blocks. The header section consists of three 80 character strings written in XDR encoded format:

File Header Section:

File Type Identifier - 80 Character String

Version Identifier - 80 Character String

Description - 80 Character String

The first string consists of the file type identifier which should correspond to the name of the PibFile class that wrote the file. This class name (with its full package path) is compared to the result of the getSamPackage function of each MECodePlugin. The file is processed by the first MECodePlugin that matches this string using that plug-in's open method.

Although the format of the remainder of the file could be written in any format that can be understood by the plug-in, it is recommended that only named blocks or "PibBlock records" be stored in the file. These records each consist of a data block header followed by the actual data values.

Data Block Header:

Block Type Identifier - 24 Character String

Block Size - Integer containing the size of the block in bytes.
(including this header)

Block Compression Flag - Integer

Block Version - Integer

It is also recommended that where possible, all plug-in specific components should implement PibBlock directly. Where this is not possible or practical it is recommended that a uniformly named method (such as "store") be added to the component that returns an appropriately configured PibBlock instance.

Please Note: PibTool, which is available at <http://www.appliedprog.com/PibTool> is a software development tool that can be used to generate source code used to create platform independent binary, (PIB) files.

2.7.2 Loading a Model

The recommended procedure for opening an MED file is as follows:

1. Create the new model.
2. Read each PibBlock
3. For each component block, create a component and add it to the model
4. Reconnect any foreign key references in component blocks by calling `reconnectIdentReferences` at the `AbstractModel` level.

The entire process is broken down into a single while loop around repeated calls to `getNextBlock`. This method retrieves the name and version information for the next block in the file. This information can then be used to determine what `PibBlock` class is required to read the next component record from the current position. Usually the block name is compared to an expected set of names to determine the appropriate class.

When loading a model, there are several CAFEAN core object types that must be handled differently: user defined numerics, view components, drawn components and annotations. The `PibBlocks` for these components and methods for loading them have been defined in the CAFEAN Core. These components, if used by a plug-in, should be loaded by the core `MEDReader` class using code similar to that shown in `PibFile Example` below.

`ViewComponents`, `DrawnComponents` and `Annotations` should be loaded using the `loadVisualComponents` method inside `MEDReader`. This method accepts a `Vector` of drawing records, a `Vector` of `ViewComponent` records and a model reference. The drawing records `Vector` should include `DrawnComponent` records (`DrawnComponentRec`), annotation records (`DrawnAnnotationRec` and `DrawnImageAnnotationRec`) and drawn user defined numerics records (`DrawnNumericRec`).

Note that `loadVisualComponents` should be called only once per model loaded and should be passed all drawing and view records.

2.7.3 Saving a Model

Model saving is handled by the plug-in's implementation of the `AbstractModel` method `saveModel`. This method handles the entire save process including file selection, overwrite prevention, access permissions checks, etc.

The recommended procedure for saving an MED file is as follows:

1. Create and the `PibFile` instance and open the file
2. Write the plug-in's package header
3. Store the plug-in's global model options
4. Store the plug-in specific components
5. Store `ModelEditor` core components

More detailed information about this process can be found in the following two examples.

2.7.4 PibFile Load/Save Example

The following is an example of a PibFile implementation for a plugin called "Example." This was not intended as fully working implementation but more as a starting point for developing a new plug-in. This example and the AbstractModel example in the next section are intended to be examined together as a model for implementing load/save logic in a code plug-in.

Plug-in specific components should be handled similarly to SomeComponent below. In this example it is assumed that SomeComponent implements the "store" method described above to return a SomeComponentRec instance. It is also assumed that SomeComponent has a constructor defined that takes a SomeComponentRec as its only parameter. In this way the component can be easily read from the MED file with only a few lines of code.

This example uses a very simplistic mapping of PibBlock names to component types. It is recommended that more sophisticated approaches be evaluated for plug-ins with a large number of components. A lookup table or standard naming scheme could be used to implement this mapping and simplify the block loading loop.

```
public class ExampleMedFile
    extends com.example.Example_file // extend the generated PibFile
{
    /** Loads a model from the given file name. */
    public AbstractModel loadModel(String fname, boolean prompt)
    {
        Vector viewBlocks = new Vector();
        Vector drawingBlocks = new Vector();
        try {
            int return_flag = OpenImportFile(fname);
            if(return_flag != 0) {
                if( return_flag == 3 || return_flag == 2 ) {
                    MainFrame.addMessage( "Incorrect format for a ModelEditor document file.",
                                           MessageWindow.UserErrorMsg );
                }
                return null;
            }
        } catch(Exception e) {
            return null;
        }
        MainFrame.addMessage( "Loading " + fname + " please wait...",
                               MessageWindow.InfoMsg);
        ExampleModel model = new ExampleModel();
        MainFrame.instance.setCurrentModel( model );

        String[] blockname = new String[1];
        int[] blockparm = new int[3];
        try {
            while(getNextBlock(blockname, blockparm)) {
                if(blockname[0].equals("EOF")) {
                    break;
                } else if(blockparm[0] < 1) {
                    MainFrame.addMessage("Error reading block["+blockname[0]+"] from "+fname+".",
                                           MessageWindow.InternalErrMsg);
                    break;
                } else if( blockname[0].equals( "UserConstantRec" ) ){
                    UserConstantRec rec = new UserConstantRec( this, blockparm );
                    UserDefinedConstant con = MEDReader.loadUserConstant( rec, model );
                    if( con == null ){
                        MainFrame.addMessage("Load failed reading user defined constant record.",
```

```

        MessageWindow.InternalErrMsg);
    } else {
        model.addComponent( con, false );
    }
} else if( blockname[0].equals( "UserVariableRec" ) ) {
    UserVariableRec rec = new UserVariableRec( this, blockparm );
    UserDefinedVariable var = MEDReader.loadUserVariable( rec, model );
    if( var == null ){
        MainFrame.addMessage("Load failed reading user defined variable record.",
            MessageWindow.InternalErrMsg);
    } else {
        model.addComponent( var, false );
    }
} else if( blockname[0].equals( "UserFunctionRec" ) ) {
    UserFunctionRec rec = new UserFunctionRec( this, blockparm );
    UserDefinedFunction func = MEDReader.loadUserFunction( rec, model );
    if( func == null ){
        MainFrame.addMessage( "Load failed reading user defined function record.",
            MessageWindow.InternalErrMsg);
    } else {
        model.addComponent( func, false );
    }
} else if( blockname[0].equals("ViewCompRec") ) {
    ViewCompRec c = new ViewCompRec(this,blockparm);
    viewBlocks.add( c );
} else if( blockname[0].startsWith("Drawn") || blockname[0].startsWith("Drawing") ) {
    // In the Example Plug-in, no records start with Drawn so this simplified
    // set of block name prefixes can be used.
    PibBlock block = MEDReader.readDrawingBlock(this, blockname[0], blockparm );
    if( block != null ) {
        drawingBlocks.add( block );
    }
} else if(blockname[0].equals("SomeComponentRec") ) {
    // Plug-in specific components follow this example with
    // each component block name in its own if case.
    SomeComponentRec someRec = new SomeComponentRec(this,blockparm);
    SomeComponent some = new SomeComponent(someRec);
    // pass false for the second parameter as the component already has an ident
    model.addComponent( some, false );
} else {
    SkipBlock(blockparm[0]);
}
}
model.validateAllComponents();
MEDReader.loadVisualComponents( drawingBlocks, viewBlocks, model );
model.reconnectIdentReferences(false, false);
model.clearDbIds();
} catch(Exception ex) {
    ex.printStackTrace();
}
return model;
}
/** Prepares the given filename to store a model. */
public boolean prepareStore(String fileName)
{
    try {
        OpenExportFile(fileName);
    } catch(Exception e) {
        MainFrame.addMessage("Failed to open file for save: " + fileName);
        e.printStackTrace();
        return false;
    }
    MainFrame.addMessage("Saving " + fileName + ", please wait...",
        MessageWindow.InfoMsg);
    return true;
}
}
}

```

2.7.5 Model Load/Save Example

The following is an example implementation of the load/save portion of AbstractModel for a plug-in called Example. Note the use of writePackageHeader to separate plug-in specific records from the core component records and the use of MEDReader utility methods to store user defined variables, constants and functions.

This plug-in uses a block called ExampleOptions as its global options object. In this case ExampleOptions implements PibBlock and thus can be written to the file directly. For some plug-ins it may be necessary to copy the global options data into a separate PibBlock instance before storing. Similar logic must be followed when storing plug-in specific components that do not implement PibBlock directly.

```
public class ExampleModel
    extends AbstractModel
{
    /**
     * This method saves this AbstractModel to an MED file using it's current save file.
     * @see #getSaveFile
     */
    public void saveModel() {
        ExampleMedFile file = null;
        try {
            file = new ExampleMedFile();
            file.prepareStore(getSaveFile().getAbsolutePath(), showProgress);
            // write the package name and PibFile name for use in opening the file later
            file.writePackageHeader( "com.example", "Example_file",
                ExamplePluginData.LABEL );
            // store the model's global options object (which is a PibBlock itself)
            ExampleOptions opts = (ExampleOptions)getModelOptions();
            opts.writeBlock(file);
            // store each component in this model
            storeComponents( file );
            MainFrame.addMessage("Save Complete.");
        } catch( Exception ex ) {
            MainFrame.addMessage("Save Failed.", MessageWindow.InternalErrMsg);
            ex.printStackTrace();
        } finally {
            file.Close(true);
        }
    }
    /** writes each component in this model to the given ExampleMedFile */
    private void storeComponents( ExampleMedFile file ) {
        Category[] categories = getFullCategories();
        for( int i = 0; i < categories.length; ++i ) {
            // views and numerics are stored below
            if( categories[i] == AbstractModel.CAT_VIEW
                || categories[i] == AbstractModel.CAT_NUMERICS ) {
                continue;
            }
            // store each of this model's plug-in specific components
            Iterator it = getComponentIterator( categories[i] );
            while( it.hasNext() ) {
                AbstractComponent comp = (AbstractComponent)it.next();
                // if the component implements PibBlock, store it directly
                if( comp instanceof PibBlock ) {
                    PibBlock block = (PibBlock)comp;
                    block.writeBlock(file, false);
                } else {
                    // if the component is not a PibBlock then custom code
                    // must be written to create a PibBlock from the component's
                    // current state.
                }
            }
        }
    }
}
```

```

    }
}
// write the ModelEditor base package header to indicate that the remainder
// of the file is core components.
file.writePackageHeader( "com.cafean.client.io.med", "MED_file", "ModelEditor" );
// store each view using the ViewComponent's store(...) method
Iterator it = getComponentIterator( AbstractModel.CAT_VIEW );
while( it.hasNext() ) {
    ((ViewComponent)it.next()).store( file );
}
// store the user defined numerics using the MEDReader utility methods
it = getComponentIterator( AbstractModel.CAT_NUMERICS );
while( it.hasNext() ) {
    AbstractComponent comp = (AbstractComponent)it.next();
    if( comp instanceof UserDefinedFunction ) {
        MEDReader.storeUserFunction( (UserDefinedFunction)comp ).writeBlock(file, false);
    } else if( comp instanceof UserDefinedConstant ) {
        MEDReader.storeUserConstant( (UserDefinedConstant)comp ).writeBlock(file, false);
    } else if( comp instanceof UserDefinedVariable ) {
        MEDReader.storeUserVariable( (UserDefinedVariable)comp ).writeBlock(file, false);
    }
}
}
}
}

```

2.8 Undo and Redo

The ModelEditor supports a single undo/redo stack that can be added to by any plug-in or operation. Undo of single modifications is supported by creating a StateEdit and adding it to the undo stack as shown below.

```
StateEdit edit = new StateEdit(component, "Single Modification");
// [ modify the component here ]
edit.end();// complete the edit
// post the undo event to the undo stack
UndoableEditEvent event = new UndoableEditEvent(this, edit);
MainFrame.instance.getUndoManager().undoableEditHappened(event);
```

Undo of multiple modifications is supported by creating multiple edits (of any kind) and appending adding them to a CompoundEdit before posting the event.

```
CompoundEdit compound = new CompoundEdit();
StateEdit edit1 = new StateEdit(component1, "Modification 1");
StateEdit edit2 = new StateEdit(component2, "Modification 2");
// [ modify component 1 here ]
// [ modify component 2 here ]
edit1.end();// complete edit 1
compound.addEdit(edit1);// add edit 1 to the compound undoable edit
edit2.end();// complete edit 2
compound.addEdit(edit2);// add edit 2 to the compound undoable edit
compound.end();// complete the compound edit
// post the undo event to the undo stack
UndoableEditEvent event = new UndoableEditEvent(this, compound);
MainFrame.instance.getUndoManager().undoableEditHappened(event);
```

Note that the above examples assume that the object(s) and/or component(s) being edited is a StateEditable object with properly implemented storeState and restoreState methods. Undo/redo for objects that are not properly StateEditable can only be accomplished by the use of plug-in specific custom undo objects.

2.9 Creating Connectible Components

A connectible component is one that can have connections between it and another component. Connections are necessary for a component when 2-way foreign key relationships are required, the relationship is to be represented in Views with a DrawnConnection, or when the connection itself has data values.

Each connectible component must keep track of the connections referring to it for editing and drawing reasons. The ConnectionList is recommended for use in holding these connection id references at runtime. See the Source Code Documentation for more information on the ConnectionList class and it's use.

In some small cases it may be enough for the connection related method to query the model for this component's connection related info. In most cases, however, the lookups in this query would be too expensive to be used in drawing code.

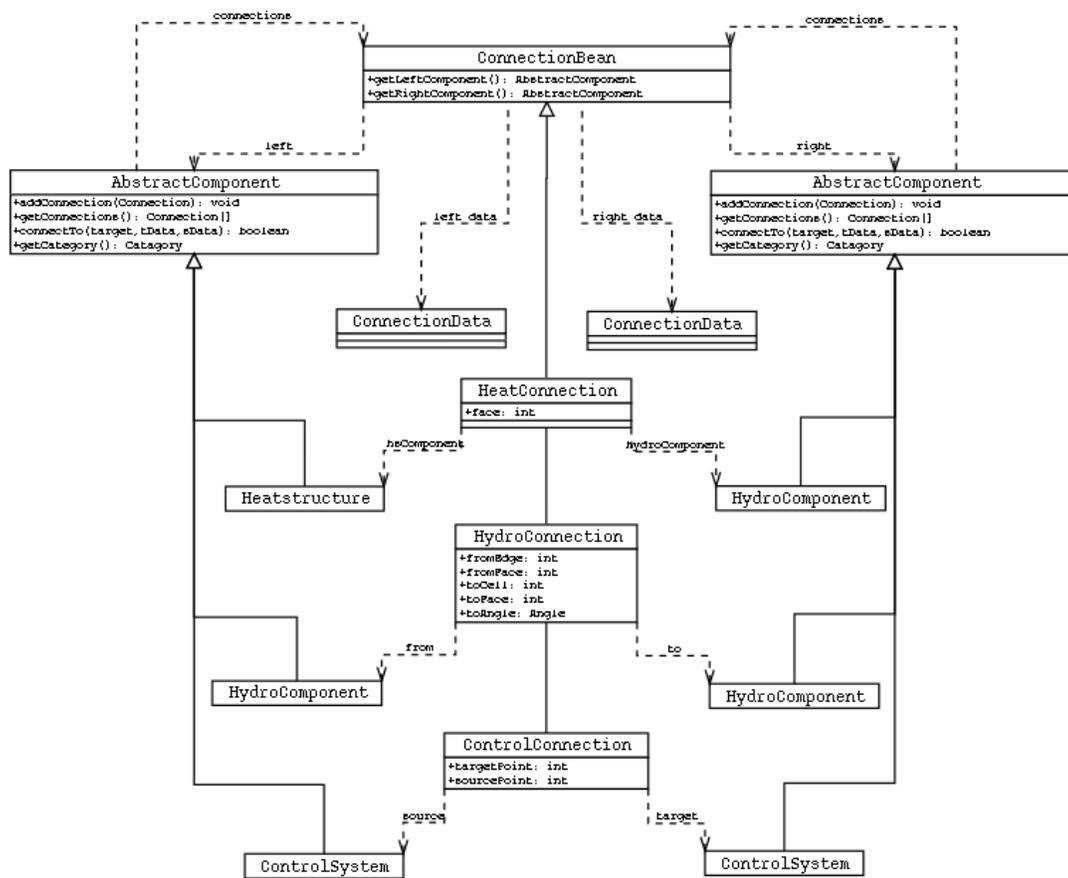


Figure 3. Connection Class UML

Figure 3 illustrates the use of Connection classes and how they connect components in the TRACE plug-in. Of the above classes, only Connection and AbstractComponent are part of the CAFEAN core. The remaining classes are presented only as an example. Additional connection types can be easily added by a plug-in by creating extensions of ConnectionBean and ConnectionData.

2.9.1 Methods to Implement

The following methods are important in the implementation of a connectible component. Additional method overrides and modifications may be necessary to implement a particular component but the following are sufficient in most cases. Refer to the Source Code Documentation of AbstractComponent for more information on available methods.

canConnectTo Method (*Required*)

This method is used to determine if this component can connect to a given component. This is used by the Connect Tool to enable or disable drop-zones on the target drawn component during a connection operation. The default implementation returns false for all components and thus must be overridden if connections are required.

getConnectionCount Method (*Required*)

This method should return the number of connections connected to this component. This method is used to dimension connection arrays and to determine if this component has connections. Normally this simply returns the size of the connection list object contained in the component or 0 if no connections are supported.

One notable exception to this is a connection such as a TRACE pipe's *jun1* reference. This connection (as well as *jun2* and *jun3*) must be treated separately from those in the connection list for very plugin-specific reasons. In this case the special connections must be counted explicitly. This sort of connection handling is more error prone and should be avoided where possible.

getConnections Method (*Required*)

This method builds an array of references to all the Connections connected to this component. The default implementation simply returns an empty array. The order of the returned array is considered arbitrary by the CAFEAN core and is assumed only to be consistent from call to call.

addConnection Method (*Required*)

This adds the given connection to the component's list of connections. The default implementation simply calls fireComponentConnected. When overriding this method, ensure that fireComponentConnected is called only once.

clearConnections Method (*Required*)

This method removes all connection references from this component. This does **not** disconnect the connections, nor does it remove them from the model. This simply clears the component's list of connections.

This method is normally used in undo and copy / paste.

disconnect Method (*Required*)

This method is used by connections to remove themselves from a given component. This method should only be called by Connection itself to ensure a one way path of disconnection. Calling this method from other places could cause an infinite loop of calls to Connection.disconnect and the component's disconnect.

createSourceData Method (*Required*)

This ensures that the given ConnectionData is suitable for a connection from this component. This is normally used by the default implementation of connectTo to create custom ConnectionData objects from SpecialConnectionData objects when connection via the Connection Tool.

GUI user interaction can be used within this method to request more detailed connection information from the user before making the connection. Returning null from this method will cancel the connection.

The default implementation of this method returns the ConnectionData passed to it.

createTargetData Method (*Required*)

This method is identical to createSourceData above with the exception that the connection in question is **to** this component.

getGroupedConnections Method (*Optional*)

This method returns all the connections to this component grouped by type with each type in a separate array. The CAFEAN pre-processor does not currently make use of this method though it is in use in several specialized plugin-specific editors.

getConnectionName Method (*Optional*)

This method is used to display connections from the perspective of this component. The default implementation returns the label of the connection and the toString of the component on the other side. The default implementation is appropriate for most situations but in some cases there may be a more appropriate name. (For example: *Inlet junction from Pipe 101*)

2.10 Plugin-Specific Unit Types

Units in the CAFEAN pre-processor are extension of Real and represent a value in SI units and it's conversion between SI and British units.

For some code plug-ins (such as CONTAIN) the analysis code will accept only SI units and the British units are available only for editing and display. For other codes (such as TRACE or RELAP5) the model may be exported in either SI or British based on the user's request. With this in mind, all code plug-ins should support both SI and British units where possible.

2.10.1 Supporting Units in the Model

Supporting plugin-specific units in the model is a matter of implementing a set of methods for mapping unit classes to their SI unit strings.

getUnitsDisplay Method (*Required*)

This method is intended for use in a selection dialog for selecting the desired unit for a particular value. The units editor for UserDefinedNumerics uses this method in particular to allow the user to choose the type of the generated value.

The returned array should include one unit per line and be in a similar form to:

< unit name > (<unit string>) or Temperature(K)

findReal Method (*Required*)

This method retrieves a new instance of the plugin-specific unit that corresponds to the given SI unit string. The given string is assumed to be identical to the string given by the unit's getSI_Units method and in most cases will be the same string.

getRealByIndex Method (*Required*)

This method retrieves a new instance of the plugin-specific unit that appears in the units display at the given index. This method is generally used directly after a unit type is chosen from the units display given by getUnitsDisplay.

getUnitIndex Method (*Required*)

This pair of methods find the units display index of the Real type or SI unit string given as a parameter. This is often used to provide an initial selection for a unit selection dialog using the units display given by getUnitsDisplay.

getDimensionless Method (*Required*)

This method should return a new instance of the plugin-specific unit that this plug-in uses for its dimensionless values. This unit is used as a default value for any unit references.

getExportUnits Method (*Optional*)

This method need only be implemented if the model must be export in a particular unit set (SI or British). Some analysis codes(such as CONTAIN) support input in only a single unit set and must be export in that set regardless of the current display units for the model.

2.10.2 Units Classes

Each unit is an extension of Real that includes a conversion factor for converting from SI to British units. The SI and British units strings are also included (such as K or m³). (Note: In some cases a m³ will be converted to m³ by the UI for readability.)

Creating each unit class is a very simple matter of extending Real, registering the appropriate editors and implementing a few methods. Because there is no additional properties needed in the extension BeanInfo classes are not needed for each unit.

In addition to implementing the following methods, each unit should register a bean editor for itself in the following manner illustrated by Energy:

```
import com.cafean.client.ui.beans.RealBeanEditor;
import com.cafean.client.ui.beans.RealArrayEditor;
static {
    // Register the defined bean editor.
    PropertyEditorManager.registerEditor( Energy.class,
                                         RealBeanEditor.class );
    // Register the defined array bean editor.
    PropertyEditorManager.registerEditor( Energy[].class,
                                         RealArrayEditor.class );
}
```

getConversionFactor Method (*Required*)

This returns the factor to multiply by to convert this unit type from SI to British. The reverse conversion (as performed by convert and getDisplayValue) is performed by dividing by the same factor.

getSI_Units Method (*Required*)

This method returns the SI unit string for this unit. For example: Temperature could be K, Length *m*, area *m*², volume *m*³.

getENG_Units Method (*Required*)

This method returns the British unit string for this unit. For example: Temperature could be *F*, Length *ft*, area *ft*², volume *ft*³.

getUnitName Method (*Optional*)

This returns the name of this unit as a single word. Often this is the same as the class name and as such the default implementation uses the base class name as the unit name.

getDisplayName Method (*Required*)

This returns the unit name in a more human readable form. This is used in various Real editors for default column names and defaults to an empty string.

2.11 Model Validation

Model validation for a plug-in is handled by the `checkModel` method in `AbstractModel` and by the use of `ValidationTests`. `checkModel` should perform a set of checks on the model and its component's current state. This method normally utilizes `AbstractComponent`'s `isOkayForExport` methods and may optionally add error and/or warning messages to the `MessageWindow`.

To use Validation Tests in a plug-in extend `ValidationTest` (overriding the methods listed below) and Implement both `getValiationTests` and `getValidationOptions` in the model. The `checkModel` implementation in `AbstractModel` will execute any available (enabled) `ValidationTests`.

2.11.1 ValidationTest Implementation

`ValidationTest` is the base class for model level validation tests that can be executed, enabled, disabled and configured. These tests are assumed to be full JavaBeans that can be edited directly in a `PropertyView`.

When creating a `ValidationTest`, the following methods must be overridden.

String getDisplayName (*Required*)

This is a short, human readable name displayed to the user when executing the test. This name does not necessarily need to be unique. Using the same display name for multiple tests is one way to separate variations of a test but hide the separation of implementation from the user.

String getShortDescription (*Required*)

This returns a long, detailed description of this validation test, its user options, and background information. This method is used by plug-ins as the pop-up help text when editing the properties of a `ValidationTest`.

boolean runValidation (*Required*)

This method is the actual validation test. If this returns false, the model will be considered to have failed. Error and warning messages should only be printed to the Message Window if the `printErrors` parameter is true. Before export this method will be called silently to determine if there are errors that require user intervention.

String getName (*Required*)

The short name of the test used when loading and storing the properties of the test. This name must be unique amongst a plug-ins validation tests.

2.11.2 ValidationTest Methods in the Model

To use ValidationTest in a plug-in, the following AbstractModel methods must be overridden.

ValidationTest[] getValidationTests (*Required*)

This retrieves the set of tests (both enabled and disabled) from the model. When called from checkModel the tests are assumed to be properly configured and ready to be executed.

ValidationOptions getValidationOptions (*Required*)

This method retrieves an options object that contains all the configured properties of the model's ValidationTests. ValidationOptions itself is a wrapper object for name/value pairs. This options object may be extended to include load and store methods for saving these options to the model's MED file.

2.12 Using the Property View

The CAFEAN pre-processor Property View is a JavaBeans™ based property editing panel. It uses Java's Introspector class and the plugin-supplied BeanInfo classes to instantiate a set of PropertyEditors for the current target beans.

No special code or configuration is required to use the Property View for editing of component or sub-component objects. By default, the ModelEditor will use the Property View for all bean-based models. Also, if the target is a ComponentElement, no special code is required to refresh the properties shown as the Property View is a ComponentListener and will be updated automatically.

The Property View bean handling has been expanded to include handling for additional optional interfaces and methods to allow more detailed customization of the UI by the target beans. The following sections explain in detail the capabilities available to plug-in authors to tailor a more responsive and organized set of PropertyEditors for a bean.

Note: Because this is a JavaBeans™ based system, to be properly represented all objects edited must conform to the JavaBeans™ architecture specification.

2.12.1 The PropertyController Interface

PropertyController is an interface describing an object that has methods to determine if its properties are currently enabled, active, required, etc. This interface is essential for the more complex components and can come in handy for even the simplest bean.

2.12.1.1 Disabled and Optional

The Property View always includes two check-boxes: one for showing *Optional* properties and the other for showing *Disabled* properties. The logic behind these two check-boxes is actually reversed from the check-box labels.

The Property View uses the method isPropertyEnabled to determine if a property is enabled or disabled and isPropertyRequired to determine if it is optional or required. These checks are made each time the view is refreshed.

To be able to support proper editing of a restart, the isRestartEditable method was added. This method is checked if the containing model is currently editing a restart.

Refer to the Source Code Documentation of PropertyController for more detailed information on the implementation of isPropertyEnabled and isPropertyRequired.

2.12.1.2 Re-sizable

Various codes have tables and arrays that can be altered but cannot be resized. For some codes this condition holds true only for editing a restart. To handle these situations the isResizable and isRestartResizable methods were added. Currently the RealArrayEditor handles these methods. New plugin-specific editors must implement support for this feature where it is appropriate.

2.12.1.3 Attribute Ordering

In most code plug-ins the relative order of properties is an important part of their presentation to the user. Since the JavaBeans™ architecture does not provide a means for ordering properties beyond the *Preferred* attribute of a *PropertyDescriptor*, this functionality has been added to the *Property View*.

To implement attribute ordering for an object the *getAttributeIndex* method must be implemented. This method returns a relative index for the property name given. Normally the returned value is the index of the given property in a statically defined array. A large integer (such as 999) should be returned in most cases for properties that have no attribute index defined.

2.12.2 Attribute Groups

Attribute Groups are a convenient way of breaking up a large number of properties into logical groupings. Each group appears as table of properties in the *Property View* with its own expansion icon and group label. Property editors are not instantiated or refreshed for groups that are not expanded.

To support Attribute Groups in a bean, a mapping between attribute names and attribute groups must be implemented using the following three static methods.

getAttributeGroups Method

This returns an array of the Attribute Group names available in this object. This is a static method and normally returns a static final array.

Note that the *General* group should not be included in this array. The *General* group is made up of those properties that do not appear in any other Attribute Group.

getAttributeGroup Method

This returns the name of the Attribute Group that the given property is part of. If the property should appear in the *General* group then null should be returned.

getAttributesForGroup Method

This method returns an array of the property names that are included in the given Attribute Group. An empty array should be returned for groups that do not exist or are empty.

Note: Returning null from this method could cause errors.

2.13 Using Registered Dialogs

A Registered Dialog is any dialog that has been added to the MainFrame's list of registered child dialogs. Registering a dialog enables the following functionality:

- Registered dialogs appear in the *Windows* menu.
- Using MainFrame's `setWindowLocation` for a registered dialog will offset the dialog's location to avoid directly overlapping (and hiding) another registered dialog.
- Dialogs registered with a model will be hidden when that model is closed.
- Registered dialogs that implement the `RefreshableDialog` interface will have their `unitsChanged` method called from `MainFrame.resetAllUnits` and their `refresh` method called after undo or redo.

In addition to the above use of the registered dialog list allows plug-ins to implement features such as:

- Ensuring that only one of a particular editing dialog is open at a time.
- Bringing a registered editing dialog to the foreground.
- etc.

addRegisteredDialog Method

This static method adds the given dialog to the list of registered dialogs. If the model parameter is not null, the dialog will be assumed to be related to the given model and will be closed when the model is closed. If the model parameter given is null the dialog will be assumed to be unrelated to any model.

getRegisteredDialogs Method

This static method returns an Iterator into an unmodifiable List of the registered dialogs. If the optional `AbstractModel` parameter is given, only those dialogs related to the given model will be retrieved.

removeRegisteredDialog Method

This static method removes the given dialog from the list of registered child dialogs. If the dialog was previously associated with a model then that model should be provided when removing the dialog from the list.

2.14 Customizing the 2D View

Current 2D View customizations include adding plug-in specific toolbars and mouse handlers as well as display element specific insertion handlers.

2.14.1 Adding Toolbars

A set of standard toolbars are available in every 2D View, regardless of the plug-in that includes:

- Main – Select, Pan, Zoom, Connect and Insert tools.
- Clipboard – Cut, Copy, Paste, Paste Special and Find.
- Annotation – Ellipse, Image, Line, Polygon, Rectangle and Text Annotations.
- Numerics – User Defined Variables and Constants.

In addition to those above, toolbars are automatically created for each of the visual parent Categories returned by the model's `getCategories` method. The buttons included on these toolbars are shortcuts buttons for the Insertion Tool. For most current plug-ins these toolbars are sufficient as most additional user interaction can be accomplished via pop-up menu items.

Plug-ins requiring additional toolbars add them using the `DrawnView` method `addToolbar`. It is recommended that toolbars be added from within the `MEPlugin` method `loadViewMenuItems`.

Note that the toolbar's *name* will be shown in the pop-up menu to show or hide the toolbar. Refer to the Source Code Documentation of `addToolbar` for more tailed information on its use.

2.14.2 Insertion Handlers and the Insertable Interface

Some visual elements require an insertion procedure that is more complex than the simple behavior provided by the Insert Tool. To customize this behavior, a more advanced Insertion Handler can be created for the element.

Currently there are two advanced handlers available:

- `RectangularInsertHandler` – Allows rectangular bounds selection before insertion. This handler is used by the Rectangular Annotation and by all Display Beans.
- `AbstractPathHandler` – Allows path point selection before insertion. This handler is used by the Line Annotation and the Polygon.

To use either of the handlers above for a visual element, simply implement the `Insertable` interface for that element and define `getNewInsertHandler` to return a new instance of the handler. So, for the rectangular handler, define `getNewInsertHandler` to return a new instance of `RectangularInsertHandler`.

To use the path handler, a subclass must be created that extends `AbstractPathHandler` and implements the following methods. Note that when creating this extension, the path

handler must maintain a reference to the element being inserted. This reference is essential when completing the insert and determining the closure point.

finishInsert

This method completes the insertion by retrieving the current path of points from the handler and setting them on the element. For a line annotation, this method simply places line points at each point and segments between them.

isClosurePoint

This method must determine if left-clicking on the given point should cause the completion of the insert. For line annotations this simply determines if the left-click is on the last point in the path.

2.14.3 Creating Custom Mouse Handlers

The mouse interaction with visual elements can be intricately controlled with `MouseListener` and `MouseMotionListener` methods implemented directly in drawn components or display beans. In some cases, however, these methods are insufficient and an entirely new tool is required. In these cases a new custom `MouseHandler` extension can be created and added to the view.

Custom mouse handlers can be added and removed from a `View` by using the `ZoomablePanel` methods `addMouseHandler` and `removeMouseHandler`. It is recommended that mouse handlers be added from within the `MEPlugin` method `loadViewMenuItems`.

The following steps are required to create a custom `MouseHandler` extension.

- Create a new class that extends `MouseHandler`
 - Override the `MouseListener` methods required (`mousePressed`, etc.)
 - Override the `MouseMotionListener` methods required (`mouseDragged`, etc.)
 - Override `activate` and `deactivate` to properly initialize and dispose of the handler's listeners and resources.
 - Override `getCurrentCursor` to return an appropriate cursor for the handler.
- Add the handler to the view's toolbar from within `loadViewMenuItems` with `ZoomablePanel`'s `addMouseHandler` method.

Refer to the [Source Code Documentation](#) for the `MouseHandler` class for more detailed information on the implementation and management of mouse handlers.

2.15 Useful Utility Classes

The following section details a list of classes and interfaces that are likely to be used in the UI portion of any code plug-in. Subsequent versions of this manual may include additional classes as they mature.

2.15.1 Interfaces

The following interfaces may be used to enable additional functionality for a particular object or editor.

Writable Interface

This interface is used primarily by the AsciiViewer to determine what components can be viewed and how to view them. It specifies a single method, write, which writes an ASCII representation of the object to the given PrintWriter.

Implementing this method for a code plug-in allows the user the convenience of being able to examine the ASCII representation of a component or object directly as it is being modified without having to export the entire model to a file between modifications.

ModelElement / ComponentElement Interfaces

These interfaces indicate an object that maintains a reference to its model and/or AbstractComponent parent. ComponentElement is a direct extension of ModelElement and is implemented by AbstractComponent.

It is recommended that all sub-components implement ComponentElement and maintain a reference to their direct parent in the reference hierarchy. This allows the CAFEAN property editing and undo architectures to properly store the state of and update the views of any objects and components being edited.

ModelDependent Interface

This interface is similar to ModelElement in that it includes an accessor for the object's model but in this case the interface is intended for use by PropertyEditors that require a model reference to edit a given value. The ComponentSelectionEditor mentioned below is a good example of a model dependent editor.

BoxSelectionListener Interface

This interface describes a listener for selection changes in a particular BeanBox. Each BeanBox has its own list of listeners accessible via addBoxSelectionListener and removeBoxSelectionListener. For more detailed information refer to the Source Code Documentation for BoxSelectionListener and BeanBox.

2.15.2 Bean Editors

The following editors are provided to simplify the creation of new code plug-ins. Special attention should be paid to the first three (ComponentSelectionEditor, RealBeanEditor and NamedIntEditor) as they are likely to be used in every code plug-in.

ComponentSelectionEditor Class

Also called an "*Ident Editor*", this ModelDependent editor is used to edit an integer that is a foreign key reference to a component. This editor includes a label displaying the toString of the target component (or *none*) and a *Select* button to choose the component ident to use.

Note that this editor is not necessarily intended to be used directly. Direct use implies that any component from any Category can be selected. This is rarely the case. Normally, extensions are created that pass a particular Category to the constructor of ComponentSelectionEditor to narrow the selection range to a given set of components. In some cases the Category may be created specifically for use in the editor extension.

RealBeanEditor Class

This editor used to edit Real values. Plugin-specific units must register themselves with this PropertyEditor directly. Refer to the section on plugin-specific units for more information on this registration.

NamedIntEditor Class

Also known as an *Enumeration Editor*, the NamedIntEditor is used to edit an integer that is an enumerated set with a description for each value. For instance: a OffOnSelEditor would edit an integer with value 0 (Off) and 1 (On). Values that fall outside of the defined range are allowed if set from outside the editor but once changed can only be reset to the original value with undo.

This editor is never used directly as the values and descriptions cannot be determined at run time. To use this editor, create an extended class that passes the possible integer values and their string descriptions to the NamedIntEditor constructor.

Note: Extensions requiring the *Namelist* functionality of the NamedIntEditor should instead extend the NamelistNamedIntEditor class described below.

Refer to the Source Code Documentation for more information on how to extend and use a NamedIntEditor.

NamelistEditor Class

This interface describes an editor for a property that conforms to the *Namelist* variable concept in which a property is actually a combination of a property and a boolean activation state. Editors of this type assume that setPropertyActive is defined in the object containing the property.

The following similarly named editors support the `NamelistEditor` interface. Refer to the Source Code Documentation for more detailed implementation details for the `NamelistEditor` interface.

NamelistIntEditor Class

This is an editor used for integers that conform to the *Namelist* variable concept as defined by `NamelistEditor`. This editor may be used directly.

NamelistBooleanEditor Class

This is an editor used for boolean values that conform to the *Namelist* variable concept as defined by `NamelistEditor`. This editor may be used directly.

NamelistRealEditor Class

This editor is an extension of `RealBeanEditor` used to edit Reals that conform to the *Namelist* variable concept as defined by `NamelistEditor`. This editor may be used directly.

NamelistNamedIntEditor Class

This is an extension of `NamedIntEditor` used for enumerated integers that can be activated and deactivated as defined by `NamelistEditor`. Like `NamedIntEditor`, this editor cannot be used directly but instead must be extended to include values and descriptions.

2.15.3 GUI Utilities

The following GUI utility classes have been used extensively in existing code plug-ins and are likely to be of use in the development of any plug-in.

TableSorter Class

This is a sorting wrapper for `TableModel` instances to allow sorting the displayed table by a user-selected column. `TableSorter` includes a method to add an appropriate mouse listener to the table header for choosing the column to sort by.

Refer to the Source Code Documentation for `TableSorter` for more detailed implementation information.

OptionPane Class

This is an encapsulation of `JOptionPane` that should be used for all option panes in the CAFEAN pre-processor. This class handles the centering of option panes on the screen rather than the `MainFrame` (if it is the parent) for single window arrangement.

3. Packaging a Plug-in

All of the class and resource files that comprise a plug-in should be placed in a jar file located in SNAP's plug-in directory. The plug-in jar file's manifest should include a `MEPluginData-Class` entry that indicates the location of the plug-in's `MEPluginData` class extension. The jar file may also include the runtime and post-processor plug-ins. For example, the manifest for the TRACE plug-in is:

```
Manifest-Version: 1.0
Plugin-Class: nrcsnap.trace.TraceCodePlugin
ClientPlugin-Class: nrcsnap.trace.TraceClientCodePlugin
MEPluginData-Class: nrcsnap.trace.TracePluginData
```

In this case, the jar file includes the preprocessor, runtime and post-processor plug-ins. The `MEPluginData-Class` entry identifies the class `nrcsnap.trace.TracePluginData` as an extension of `MEPluginData`.

4. Preprocessor Python Scripting

Scripting support in the preprocessor client is available via a Python interpreter. Scripts can be run from batch mode with the MACRO batch command. Using this scripting interface gives the user direct access to the internal structures of the ModelEditor. It is important to note that in some cases this direct access may have unintended affects on the structures being accessed.

A discussion of the facilities and classes available to support scripting is provided below along with several examples. Though some examples in this document use the TRACE plug-in, the methodology should apply equally well to any JavaBean™ based plug-in.

4.1 Built-in Python Methods

The following Python methods have been provided to assist in accessing desired components and outputting messages to the user:

- `addMessage(message)`

This method is used to print messages to the user as Message Window notices. Note that multiple line messages in the Message Window may not appear properly. In these cases it is recommended that multiple `addMessage` calls be used.

Example:

```
addMessage("Script beginning.")
```

- `addError(message)`

This method is used to print messages to the user as Message Window internal errors. As with `addMessage`, multiple line messages in the Message Window may not appear properly and should use multiple `addError` calls instead.

Example:

```
addError("Script failed.")
```

- `getModel()`

This method is used to retrieve the current model for the script executing. The current model in batch mode is either the last model imported/opened or the model specified in the MACRO batch command.

Example:

```
addMessage("Script running with model: %s"\n           %(getModel().getName()) )
```

- `findComponent(catName, number)`

This method is used to retrieve a given component by its component number and category name from the script's current model. The category names are identical to those used by the Navigator. `findComponent` uses `getModel` internally to call `findComponentByCC` on the current model. Scripts which deal with multiple models simultaneously will need to call `findComponentByCC` directly for each component required.

Example:

```
addMessage("Found pipe: %s"\n           %(findComponent("Pipes", 2).toString()) )
```

4.2 Core CAFEAN Classes

Familiarity with parts of the following core CAFEAN classes is important when writing more advanced Python scripts. As a minimum the user should be familiar with Real class presented below.

4.2.1 Real Class

Real is a base class for all floating point numbers in the ModelEditor. It's main purpose is to centralize the handling of unit types and conversions for it's subclasses and to allow the display and editing of values in either SI or British based on the user's current preference. Each plug-in will have its own set of Real derivatives used to display various types of values (such as length or temperature). Currently, the CAFEAN core has only Time(seconds) and Angle(degrees) available as examples. Refer to the programmer's documentation for each plug-in for more information on what unit types are available.

From a scripting perspective, the important methods to note for Real are:

- `getDoubleValue()`
This method returns the current SI value of the Real. All floating point values are stored as their SI value and converted to British only when requested by the model for display.
- `toString()`
This method returns a formatted string representation of the Real's current value in the model's current units(SI/British).
- `toString(unitType)`
This method returns a formatted string representation of the Real's current value using the given unit type as either `Real.SI` or `Real.BRITISH`.
- `getDisplayValue(unitType)`
This method returns the double value of the Real in the requested unit type. Unit type is either `Real.SI` or `Real.BRITISH`. Note that any calculations performed with Real values that assume British units requires that `getDisplayValue` be used in place of `getDoubleValue`.

4.2.2 AbstractComponent Abstract Class

AbstractComponent is the base class for all *Components* in the ModelEditor. Objects such as TRACE's Pipes, Tees and Control Systems are all AbstractComponent derivatives. Though nearly all script interaction with AbstractComponents will use component-specific properties it is important to note the few things they have in common:

- `GetComponentNumber()`
This method returns the component's *number*. What this number means can be very

plug-in and component specific. For most components this amounts to the component's unique identifier in that model. In the TRACE plug-in this actually corresponds to the component number input.

- `label()`
This method returns a string describing the type of the component. For a TRACE pipe, “Pipe” is returned. This is usually similar but not necessarily the same as the name of the Category for the component.
- `toString()`
This returns a human-readable string representation of the component. This is usually of the form: “<label> <component number> (<name>)” but can differ significantly between component types. In most cases this is also the string displayed in the Navigator node for the component.
- `getCategory()`
This returns the Category that the component is part of. This category can be used in component lookups in the model (in place of the category name) or simply to display the general grouping of a particular component.

4.3 TRACE Plug-in Examples

Below is a brief description of the general types of TRACE components, followed by a set of examples, explanations and special cases that illustrate the use of the ModelEditor's Python scripting capability within the context of the TRACE plug-in. It is important to note that in general the TRACE plug-in variable naming convention matches the TRACE input manual. Also, the accessor methods for all variables conform to the Java™ language standard for JavaBean™ method names. Some exceptions to the TRACE naming convention were required to handle special cases of input constraints (foreign keys) and data organization (vessel axial levels). Refer to the programmer's documentation for the TRACE plug-in for specific information on available values and their data types.

4.3.1 Hydraulic Components

Hydraulic components fall into three general groupings: boundary condition components such as Break or Fill, fluid components such as Pipe or Tee and the 3D Vessel. Each of these groupings has a slightly different internal structure and must be treated differently. As always, refer to the programmer's documentation for more specifics.

Boundary Components (Fill and Break) are the simplest of general groupings as they contain only component-level data. The majority of the values contained in these components will match the input manual names exactly. The following examples show the retrieval of various values from a Fill component.

```
fill = findComponent("Fills",8)
addMessage(" *          jun1          ifty          ioff")
addMessage("%13s %13d $13d\"
           # the junction number to which the FILL is connected
           %(fill.getJun1CC(),\
           # FILL type
           fill.getIfty(),\
           # FILL fluid state option
           fill.getIoff()))
```

Fluid components are hydraulic components that have one or more fluid segments. A fluid segment is a collection of cell and edge data and is normally referred to as *main tube* or *side tube* in the TRACE input manual. In many cases the main tube is assumed for the discussion of elements of a fluid component with a single fluid segment. Note that in the TRACE plug-in there is no *phantom cell*. All phantom cell compensation is handled in the ASCII export code.

The structure of the following classes is especially important when handling the data of a fluid component:

- **FluidComponent**

This class is the base class for all fluid type components. It contains accessors for retrieving it's FluidSegments, Cells(`getCellAt`), and Edges(`findEdgeAt`). It is

important to note that when 2 fluid components are connected, they share edge data for the edge that is connected. In the TRACE input format, this data is entered for both edges and assumed to be equivalent. To ensure that the proper edge data is being retrieved the `findEdgeAt` method must be used instead of `getEdgeAt`. Also, because of this sharing of data, some values (such as `fric` and `grav`) will appear to be inverted if two inlets or two outlets are connected. Another important set of methods are used to retrieve the junction numbers for the inlet, outlet (and side) junctions of a fluid component. `getJun1CC`, `getJun2CC` and `getJun3CC` retrieve the inlet, outlet and side tube junction numbers respectively.

- **FluidSegment**

A FluidSegment is one tube of a hydraulic component. A Tee component contains two FluidSegment, one main tube, one side tube. Each segment contains an array of Cells and Edges that can be retrieved with `getCells` and `getEdges` respectively. In addition, FluidSegments contain some fluid power. Refer to the programmer's documentation for more information on additional FluidSegment variables.

4.3.1.1 Example Scripts

The two most important examples of fluid components are Pipe and Tee. The following examples show the retrieval of various values from each type.

Example 1: Pipe Component General Data Access.

```
# An example of general pipe data access
pipe = findComponent("Pipes",2)
addMessage( "*" ncells nodes jun1 jun2 epsw" )
addMessage( " %d %d %d %d %f" \
            %( pipe.getFluidSegment(0).getCellsCount(), \
              pipe.getNodes(), \
              pipe.getJun1CC(), pipe.getJun2CC(), \
              # Note the use of getDoubleValue()
              pipe.getEpsw().getDoubleValue() ) )
addMessage( "*" ncells nodes jun1 jun2 epsw" )
addMessage( " %d %d %d %d %f" \
            %( pipe.getFluidSegment(0).getCellsCount(), \
              pipe.getNodes(), \
              pipe.getJun1CC(), pipe.getJun2CC(), \
              # Note the use of getDoubleValue()
              pipe.getEpsw().getDoubleValue() ) )
```

Example 2: Calculation of Total Volume for a Pipe Component .

```
# This is an example of retrieving data directly from a pipe's cells in
# order to calculate the pipe's total volume.
totalVolume = 0.0 # initial value of 0.0
idx = 0           # Current cell index.
                 # Note that all indexes are in C notation {0 to (n-1)}
while idx < pipe.getCellCount():
# getCellCount is the total number of cells in the component.
  cell = pipe.getCellAt(idx)
```

```

    totalVolume += cell.getVol().getDoubleValue()
    idx += 1 # next cell
addMessage( "Total Volume: %1.3f m^3" % totalVolume )

```

Example 3: Display of Edge Data for Pipe Component.

```

# This example shows retrieving data from a pipe's edges for
# displaying each flow area
idx = 0 # Current edge index.
addMessage( "%4s %14s %14s %14s" \
            %("edge", "flow area", "hyd diam", "grav"))
# getEdgeCount is the total number of edges
while idx < pipe.getEdgeCount():
    edge = pipe.findEdgeAt(idx)
    addMessage( "%4d %14s %14s %14f" % \
                ( (idx+1), \
                  edge.getFa().toString(), \
                  edge.getHd().toString(), \
                  edge.getGrav() ) )
    idx += 1 # next edge

```

Example 4: Calculation of Total Volume for a Tee Component .

```

# This example shows retrieving volumes from a tee side-tube to
# calculate the total volume.
ncells1 = tee.getFluidSegment(0).getCellsCount()
ncells2 = tee.getFluidSegment(1).getCellsCount()
idx = 0 # Current cell index.
while idx < ncells2:
    # offset by ncells1 to get the proper index
    cell = tee.getCellAt(ncells1 + idx)
    totalLength += cell.getDx().getDoubleValue()
    totalVolume += cell.getVol().getDoubleValue()
    idx += 1 # next cell

```

4.3.1.2 3D Vessel Component

The 3D Vessel component contains an array of VesselLayers, each with an array of Cells and 3 arrays of Edges (1 per axis). The recommended method for retrieving Cells from a Vessel component is `getCellAt(z,p)`, where `z` is the axial level and `p` is the planar cell index. Edges are similarly retrieved with `getEdgeAt(p,z,face)` where `z` and `p` are axial level and planar cell respectively. The remaining Vessel data closely follows the TRACE input manual for naming. Refer to the programmer's documentation for more information on Vessel component data structures.

The following example shows how to retrieve and display the hydraulic diameter for each cell of a Vessel.

Example 5: Vessel Component General Data Access.

```

nasx = vessel.getNasx() # Number of axial vessel levels
planars = vessel.getNrsx() * vessel.getNtsx(); # number of planar cells
addMessage("%5s %6s %14s %14s %14s" % \
            ("axial", "planar", "hdxr", "hdyt", "hdz"))

```

```

level = 0 # current axial level
while level < nax:
    planar = 0 # current planar cell
    while planar < planars:
        addMessage( "%5d %6d %14s %14s %14s"% \
                    ( (level+1), (planar+1), \
                      vessel.getEdgeAt(planar,level,0).getHd().toString(), \
                      vessel.getEdgeAt(planar,level,1).getHd().toString(), \
                      vessel.getEdgeAt(planar,level,2).getHd().toString())
        planar += 1 # next planar cell
    level += 1 # next axial level

```

4.3.1.3 Heat Structures

The properties of note for a Heat Structure are an array of HeatCells (`cells`), an array of SupplementalRods (`nhot`) and the MeshpointTable (`mesh`). Each HeatCell has an outer and inner Surface that contains the boundary condition data for the cell. The SupplementalRods contain the initial temperatures (`rftn`) and the fuel burnup (`burn`) array. Additional upper and lower boundary temperatures are stored in the heat structure (average rod) and in each SupplementalRod. The remaining data closely follows that specified in the TRACE input manual. Refer to the TRACE plug-in programmer's documentation for further information.

Example 6: Heat Structure General Data Access.

```

# This block shows an example of retrieving heat cell length
# and surface heat flux for the inner and outer surfaces.
nzhstr = htstr.getCellsCount() # number of axial heat cells
nodes = htstr.getNodes()      # number of radial nodes

addMessage("Cell Length and Inner Surface Heat Flux")
addMessage( "%6s %14s %14s" %("cell","length","inner flux") )
cell = 0 # current axial heat cell
while cell < nzhstr:
    c = htstr.getCells(cell)
    addMessage( "%6s %14s %14s" \
                %( cell+1, \
                  c.getDhtstrz().toString(), \
                  c.getInner().getQflxbc().toString() ) )
    cell += 1

```

4.3.2 Included files

The following files have been included as more extensive examples of data access for the TRACE plug-in. Each of these examples has been tested to ensure correct syntax and can be executed with the accompanying Python.batch file.

- fill.py

This file contains examples that apply well to both Fill and Break components. The variables used are specific to Fill.

- `htstr.py`

This file contains various examples of data access for a Heat Structure component. This includes surface heat flux and initial temperature array access for the average rod.
- `pipe.py`

This file contains examples of fluid component related access. Cell and edge data access is shown for use in calculating total cell lengths/volumes and retrieving the flow areas and hydraulic diameters of edges.
- `tee.py`

This file contains additional fluid component examples that deal with access to the second fluid segment.
- `vessel.py`

This file contains examples of access to vessel cells and edges by axial level and planar cell.
- `trace_util.py`

This file contains an unfinished set of python wrappers for hydraulic components that provides an example of more advanced uses of the existing scripting functionality.