

Architectural Overview

This document provides a high-level overview of the various classes which together comprise the Sheets Hypercode Environment. While it cannot claim to be comprehensive or even complete, it should give enough grounding in the basic concepts to allow informed browsing of the code itself. Ultimately, of course, the code is the definitive description of the system.

If you have questions about the code and architecture which are not covered by this document, feel free to ask for clarification. For the foreseeable future, many of the original authors will be reachable via gwydion-bugs@cs.cmu.edu.

Caveat: The Sheets project strongly advocates evolutionary design of software systems, and this code reflects this in two different ways. Firstly, we have striven to produce an environment which simplifies understanding and extension of complex systems, even when they have not been created via formal design techniques. Secondly, Sheets has itself evolved from a small demonstration project into a full-featured Java environment, and has gradually evolved from JDK1.0 into JDK1.1 (and more recently JDK1.4). This has resulted in a far more powerful system than we could have ever envisioned in a single design phase, but has also resulted in code whose organization is less than perfect. In addition, 95% of Sheets was coded using Sheets. For better or worse, this has allowed us to easily create, maintain, and extend classes which might be quite unwieldy when viewed from a lesser environment. (We don't actually advocate development of 1000 line classes, but Sheets has no problem making sense of them.)

1 Fundamental Concepts

1.1 Fragment

The [Fragment](#) class comprises the basic data model for the Sheets environment. Each fragment is a persistent [ArchivalObject](#) which represents some user-level program entity. The user may view and manipulate them through [Viewers](#), or create new ones via [LanguageExperts](#).

The exact granularity of Fragments have been chosen on a somewhat ad-hoc basis, but they obey one basic principle: each fragment should be as small as possible while still representing an independently meaningful object. For example, we find it reasonable to divide Java code into distinct methods and fields, but not to further divide into statements and

expressions. Similarly, documentation is divided into sections and paragraphs, but not into individual sentences.

The fragment model embraces a fundamental dichotomy. They are typically viewed and edited as text, and can even be changed in fairly arbitrary manners. On the other hand, they are still structured objects, which can store arbitrary internal state that need not be displayed to the user. This hidden information can be selectively revealed to the user as *attributes* or via alternative *views* or simply maintained as internal state to support other operations.

Given the fundamental nature of Fragments, they have a fairly complex extension protocol. However, there are a small core of methods which all subclasses of Fragment are expected to implement:

Methods which create new Fragments must make sure they are *registered* with [SentinelMgr](#) via [brandNewFragment](#). Note however, that the [Replacement](#) mechanism automatically registers all fragments passed to [replaceWith](#), so nothing need be done for replacements.

1.1.1 Viewing

Each fragment supports a number of different *views* and is responsible for generating [Viewers](#) to display the fragment in accordance with these views. In order to support common viewing modes (and auxiliary functions such as searching) they are also responsible for generating Strings representing a set of *standard views*.

Most fragments will support viewing by extending the methods [standardViews](#), [makeView](#), and [toString](#).

1.1.2 Replacement

Although fragments may be modified by changing attributes or (for *container* fragments) by changing their contents, most editing operations end up instead *replacing* a fragment with one or more new (but typically related) fragments. Although this seems counter-intuitive, it is what allows us to split one paragraph into two by putting blank lines in the middle, or to convert a Java method into a comment by adding `"/"` at the beginning of each line.

Most fragments will support editing and replacement by overriding the [replaceWith](#) method. This routine takes the string produced by a user's edit and converts it into a set of fragments. It should then call a different lower-level [replaceWith](#) method to perform the actual replacement.

These fragments may also wish to extend [pickSuccessor](#) to designate a *primary* replacement, or the [sameInterface](#) and [affectedBy](#) methods so that the user can be notified of any potentially global effects the replacement might have.

Architectural Overview

Fragments which support the "extend-fragment" command or auto-extension should implement [newExtension](#) and [isEmptyExtension](#).

1.1.3 Attributes

Each fragment can define arbitrary "out-of-band" data attributes which the user can view and manipulate. These can either be *scalar* or *list-valued* attributes. Scalar attributes are integral parts of the Fragment itself (such as the package name for a Java fragment), and are typically viewed and manipulated as text. List-valued attributes contain references to an arbitrary number of other self-contained fragments. They may be *constrained* to contain Fragments which satisfy certain properties.

Attributes are specified via the [supportedAttributes](#) method, which returns a Vector of [Attribute](#) objects. Fragments which are created by replacement can copy attributes from the original via the [inheritAttributes](#) method, while brand-new fragments can encourage users to specify values by listing them in [attributesShownOnCreation](#).

Although any attribute of a fragment can be retrieved via the supportedAttributes list, the [findListAttr](#) method can provide a potentially faster shortcut for retrieving list-valued attributes.

1.1.4 Persistence

Sheets provides two different mechanisms for making Fragments persistent. The load/store mechanism writes binary data to the active database, while the load/save mechanism *dumps* data to an ASCII interchange format which provides a backup and DB evolution tool.

The first mechanism is based upon the [PObject](#) interface, and is typically extended via the [loadData](#) and [storeData](#) methods. Persistent members will be manipulated via [DB.lookupID](#) and [getID](#) methods. The second method is based upon the [ArchivalObject](#) class and is typically extended via [getArchivalType](#), [saveCoreInfo](#), [saveAuxiliaryInfo](#), [loadAuxiliaryInfo](#), [finalizeLoad](#), and [isDumped](#).

1.1.5 Notification

Whenever something happens to a Fragment, be it replacement, modification, or simply a change in status, Sheets *notifies* various other objects by passing them appropriate *events*. Sentinels, Viewers, and other PObservers can all receive [FragmentEvents](#) when fragments are replaced, changed, or edited. Others may receive [ContainerEvents](#) when fragments are added, removed, moved, or replaced within [FragmentLists](#).

In addition, Sentinels can receive [AscensionEvents](#) or [DescentEvents](#) whenever any operation changes a fragment's *plane of existence*. The [planeOfExistence](#) denotes the fragment's

reachability and how it is referenced. This indicates whether the fragment is semantically part of the program, or is dead, or whatever. Possible values are [IN THE PROGRAM](#), [HANGING OUT](#), [GONE BUT NOT FORGOTTEN](#), or [WAY TOTALLY DEAD](#).

1.2 Attribute

Instances of the [Attribute](#) class are used to provide controlled access and viewing of various interesting aspects of a [Fragment](#). [ScalarAttributes](#) provide access to various parts of the Fragment, while [ListAttributes](#) describe arbitrary lists of other Fragments that are somehow associated with the Fragment. Note that Attributes objects don't actually contain values -- they provide access to the values stored in the Fragment.

Each Attribute is created with a name and an associated Fragment, which may be accessed via [getName](#) and [getFragment](#). Optionally, they may cause attribute flags to be shown in the sidebar by extending [isInteresting](#), or prevent modification of the attribute value by extending [isWritable](#). (Note that you typically achieve the latter by simply inheriting from [StaticScalarAttribute](#).)

The [ScalarAttribute](#) class provides the ability to display and potentially modify a string describing some aspect of the Fragment. You must override [getValue](#) to retrieve the value from the Fragment and may override [setValue](#) to modify it when the user specifies a new value. Note that `setValue` can't be called directly, but can be called indirectly via [changeValue](#).

The [ListAttribute](#) class provides capabilities to retrieve and display named [FragmentLists](#) associated with the Fragment, or to create them if they don't yet exist. You can often use this class directly without subclassing it. However, you can still extend it to provide specialized behaviors. Most commonly, you will extend [makeConstraint](#) to force any newly created [FragmentLists](#) to constrain their values via a [ListConstraint](#). You may also override [defaultView](#) to specify how the contents of the [FragmentList](#) should be viewed by default or [isUnordered](#) to specify that two lists can be considered effectively identical even if they are in different orders. Very occasionally, you will need to extend [makeList](#) to provide special actions when a new [FragmentList](#) has to be created.

1.3 Sheet

The [Sheet](#) class embodies the most common variety of collecting [Fragments](#) together. It contains a String *value* and a [FragmentList](#) which comprises its *contents*. Sheets are most commonly displayed and modified through a [SheetViewer](#), which gives a file-like textual view, or a [TOCSheetViewer](#) which gives a compact, narrow summary suitable for quick navigation.

1.4 FragmentList

As its name implies, the [FragmentList](#) class manages ordered sequences of [Fragments](#). This list may comprise the contents of a [Sheet](#) or the value of a list-valued [Attribute](#). The sequence of fragments be explicitly modified (and displayed) via a [ListViewer](#) or automatically maintained via a [ListConstraint](#).

Each FragmentList has various simple properties. The [name](#) field describes its function, while [getOwner](#) returns a unique Fragment which "owns" the list. The list is regarded as "interesting" if the user has explicitly modified it in some way, and may be forced to contain Fragments fitting some arbitrary description by associating a ListConstraint. The appropriate methods are [isInteresting](#), [makeInteresting](#), [getConstraint](#), and [setConstraint](#).

The Fragments contained within the list may be accessed via [numFragments](#), [getFragment](#), and [getFragments](#). The contents may be changed via a wide variety of methods which are, for the most part, self-explanatory: [addFragment](#), [addFragments](#), [replaceFragment](#), [removeFragment](#), and [moveFragment](#).

A FragmentList may be "frozen". This is a feature intended for use in version control, and forces the list to contain specific version of Fragments, even if they are later modified, replaced or deleted. The methods [isFrozen](#), [setFrozen](#), [getRawFragments](#), and [getActualFragment](#).

1.5 SheetViewer

The [AbstractSheetViewer](#) class is a ViewContainer which optionally shows the value a [Sheet](#), and can be *opened* to show the contents in a [ListViewer](#). (Like most instances of [Viewer](#), it can also show [Attributes](#) of the Sheet.) Usually, you will use a the [SheetViewer](#) subclass, which will show the contents in arbitrary views specified by the user. The special [TOCSheetViewer](#) subclass is restricted to showing "toc" views for the sake of compactness.

1.6 Viewer

The [Viewer](#) class is probably the most complex class in the Sheets system, since it serves as the primary interface between the user and the Fragments which make up his program. It serves as the "view" portion of the MVC paradigm, and as such interacts closely with the [Fragment](#) class (which it is viewing) and the [ViewPanel](#) class (which provides high-level control). It also handles some control functions which are delegated by the ViewPanel.

Viewers closely mimic AWT components, but do not inherit from the Component class. This is because we needed to have full double-buffering support and found that early versions of AWT could not perform proper buffering on nested Components. It is unclear whether the

mbination of lightweight components and Swing would now suffice, but we are in no rush to retro-fit the code, especially in light of Swing's potential performance problems.

Viewers serve many different functions: layout, display, view customization, editing, mouse-event handling, attribute display, and context-dependent help.

1.6.1 Layout and Display

Most Viewers can support window layout by simply extending the [baseHeight](#) method so that enclosing ViewContainers know how much vertical space to allocate for them. When any operations change this height, they should call [invalidate](#) so that the layout can be recomputed. Viewers which wish to display non-standard information in the *sidebar* should consider extending [controlBlockHeight](#), while viewers which contain sub-views may need to extend [contentHeight](#).

In order to display themselves in the window, most Viewers need to extend the [paintContents](#) method, which should paint the "base" value and recursively paint any "contents", but need not concern itself with attributes or highlighting. Some special Viewers may wish to implement non-standard behavior by overriding [highlightSelection](#), [paintBackground](#), or [drawControlBlock](#). You should only override [paint](#) if you wish to drastically change the way a fragment is displayed or if you need to do some special preparation before the fragment is painted.

1.6.2 View Customization

Viewers adjust their appearances according to several different states. They have a *view* which controls the gross appearance or verbosity of the display, and any number of *view modifiers* which permit minor customizations. Many viewers tend to support the standard views "header" and "full", while other standard views such as "toc" and "summary" are typically handled by special-purpose Viewers.

Each viewer should extend [getView](#) to return the name of the current view, and extend [setView](#) to change to a potentially different view. Note that [setView](#) is allowed to simply call the superclass method which will create a new Viewer to handle the specified view.

Modifiers are handled by a suite of functions which specify possible modifiers and their status and which allow the modifiers to be changed. The former include [supportedModifiers](#) and [showingWith](#), while the latter include [showWith](#) and [showWithout](#).

1.6.3 Editing

For the most part, both display and editing is handled automatically by inheriting from TextualViewer or RichTextViewer. If you inherit from the latter, you may wish to extend

Architectural Overview

[RichTextViewer.restyleLines](#) so that the styling can be updated whenever the text changes. Some viewers might also wish to extend [TextualViewer.desiredIndentation](#) to handle automatic indentation or [TextualViewer.fragmentComplete](#) to support *auto-commit*.

1.6.4 Mouse Event Handling

Although [ViewPanel](#) is responsible for handling all windowing events, mouse clicks are typically passed on to the containing Viewer for interpretation. The methods [leftClick](#) and [doubleClick](#) tend to perform selection actions, and can usually simply be inherited from Viewer. [rightClick](#) and [clickControlBlock](#) are used to pop up context-sensitive and view-modification menus. You probably needn't override them, but you will likely want to extend [fillContextMenu](#) to include any commands specific to this Viewer or the Fragments it views. [middleClick](#) typically implements a smart "goto-word" capability, and is optional.

1.6.5 Attribute Display

Attribute display, editing, and modification are all handled automatically by the Viewer class. You only need extend any methods if you wish to prevent attribute display or editing for a certain Viewer class, in which case you should override [hasAttributes](#) or [canBeEdited](#).

1.6.6 Context help

Context help, like the middle-click "goto-word" capability is optional, but may be worth supporting for Viewers with strong semantic knowledge. You do this by overriding [showContext](#) with a function that examines the selection and calls [newContextInfo](#) with information about that selection.

1.7 ListViewer

The [ListViewer](#) class is another major workhorse class. It is the primary class used for viewing and manipulating [FragmentLists](#) (whether they occur as [Sheet](#) contents or list-valued [Attributes](#)). It is both a [Viewer](#) and a [ViewContainer](#) -- it holds the subviews for all of the Fragments contained within the [FragmentList](#) and thus handles layout and delegation for each of the contained Viewers.

ListViewers always lay out their subviews in the default vertical tiling scheme favored by [Sheets](#). More exotic layouts for [Graphs](#) or [Tables](#) are handled by other [ViewContainer](#) classes.

1.8 Query

The [Query](#) class implements a framework for performing queries -- i.e. finding a complete list of [Fragments](#) which match a certain arbitrary criterion. Queries are typically added to the

QueryDialog by an appropriate [LanguageExpert](#). Queries may operate by searching through the entire set of Fragments known to Sheets, or they may accelerate their searches by using information gathered by associated [Sentinels](#). They may also supply [ListConstraints](#) for any [Sheet](#) created from the query results.

Each subclass should implement one or both of the methods [fitsQuery](#) and [quickSearch](#). The Query class will then perform a quick search to find all potential matches, possibly weed out any fragments that don't appear on a particular target [Sheet](#), and finally call fitsQuery on any fragments that remain.

Subclasses should override the toString method to provide a plain English description of the query. They can also suggest a string for search highlighting by overriding [whatToHighlight](#), return the results in some preferred order by overriding [sortResults](#), or provide a ListConstraint for any associated Sheet by overriding [createConstraint](#). (If createConstraint returns a [GraphableConstraint](#), you should also override [graphIsUsable](#) method to return true.)

After creating a constraint, callers can use [getQueryResults](#) to retrieve a Vector of results.

2 Support Classes

2.1 ViewContainer

The [ViewContainer](#) interface describes a set of operations supplied by any class which can contain nested [Viewers](#). This includes facilities for redrawing, recentering, navigating through, or updating the layout for the subviews, replacing Viewers with alternate versions, and specifying the default display characteristics for subviews. Note that, because of the potential for displaying attributes, all Viewers are also ViewContainers.

2.2 ViewPanel

The [ViewPanel](#) class is an AWT Component which displays the contents of a [Sheet](#) or other ContainerFragment. As such, it anchors an arbitrary hierarchy of [Viewers](#), and serves as an intermediary between the true AWT Components comprising windows, scrollbars, etc. and the lightweight imitations represented by fragment Viewers. ViewPanels may be accessed from any Viewer via the [Viewer.getViewPanel](#) method.

ViewPanels handle several sorts of responsibilities: double-buffered display, scrolling, event handling, and selection management.

2.2.1 Display and Scrolling

The most common display-management operation is [update](#), which takes responsibility for

Architectural Overview

making sure that all contained Viewers are properly formatted and then redraws the screen. Other display functions are handled by an inner class named [Painter](#), whose single instance is stored in the [painter](#) field. This class mostly operates behind the scenes, but is responsible for maintaining the double-buffer canvas, tracking the size of the panel, and calling appropriate paint routines on the contained Viewers. Most external callers will only be interested in [forceUpdate](#), which forces an immediate repaint of the window rather than queuing it for eventual painting, and [getMetrics](#) which grabs the font metrics used for character painting. (We recommend using this routine rather than calling `getFontMetrics` directly, since the latter can sometimes produce unreliable results under JDK1.2.)

Scrolling is handled by the inner class [Position](#), whose single instance is stored in the [position](#) field. It is responsible for updating and responding to the scroll bar and through the [getFirstLine](#) method, tells the painting routines which portion of the contained Viewer should be painted. Other classes can affect the positioning directly by calling [doLineUp](#), [doLineDown](#), [doPageUp](#), and [doPageDown](#), or can make sure that desired locations are visible by calling [recenter](#) or [recenterCursor](#).

2.2.2 Event Handling

ViewPanels intercept all AWT events, and can thus be considered to be the "Controller" portion of the MVC paradigm. However, some events are simply passed on to Viewers for specialized interpretation. Event handling is handled by the inner class [Controller](#), whose single instance is stored in the [controller](#) field. For the most part, this class can be considered a black-box, but you may find yourself wishing to set the [goingToContext](#), [middleClick](#), or [openingSheet](#) fields. Setting these fields trigger certain actions when the mouse-button is released, and thus work around a bug in some JavaVMs that causes very strange results when windows are created while the mouse-button is still pressed.

Most events handled by the Controller end up either setting the selection (described below) or executing `EditCommands`. Keystrokes are generally converted directly into editing actions, while mouse clicks are passed on to the selected Viewers which will determine and execute an appropriate action. Viewers can customize behavior by specializing the methods [leftClick](#), [middleClick](#), [rightClick](#), [doubleClick](#), and [clickControlBlock](#).

2.2.3 Selection Management

Each ViewPanel maintains a Selection which denotes either multiple contained Viewers or a portion of a single contained Viewer. This selection is queried by calling [getSelection](#) and [hasMultiSelection](#), and updated by calling [setSelection](#) or by calling methods on the Selection object returned by `getSelection`. Further information can be obtained from [selectedSheet](#), [selectedList](#), and [selectedContainer](#), which seek to find an appropriate object which contains the current selection. There are also a variety of operations which perform

some operation upon the current selection and which were placed in ViewPanel for lack of a better place. These include [removeSelectedFragment](#), [destroySelectedFragment](#), [cutSelectedFragment](#), [pasteFragment](#), [copySelectedFragment](#), [copySelection](#), [cutSelection](#), and [deleteSelection](#).

2.3 LanguageExpert

The [LanguageExpert](#) class attempts to act as the sole point of contact between a family of [Fragment](#) classes and the rest of Sheets. It serves as a delegate for many different subsystems, including fragment creation, persistence, viewing, queries, commands, sentinels, importation, exportation. While this seems like a great deal, it should have the happy effect of allowing a new family of Fragments to be added to the system by simply adding a single statement to [LanguageExpert.initializeAllExperts](#).

2.3.1 Persistence

Each of the two persistence mechanisms supported by Sheets require an arbiter to translate an arbitrary object/type token into a set of actions for loading an actual object. The "dump/undump" mechanism relies upon [persistenceTypes](#) to specify a set of tag strings which are understood by the expert and [loadFragment](#) to call appropriate loading routines for any dumped fragment which is labelled by one of those tags. The "load/store" mechanism relies upon [loadableClasses](#) to provide a complete list of [PObject](#) classes associated with the family of Fragments.

2.3.2 Viewing

The [defaultViewModifiers](#) method specifies a list of *view modifier* strings which are applicable for fragments controlled by the expert. It needn't list all such modifiers -- just the ones which should be turned on by default.

2.3.3 Queries

The [makeSomeQueryPanels](#) method creates a set of [QueryPanels](#) which can be plugged in to the general [QueryDialog](#) menu. Each of these panels collects data with which to create a [Query](#) object which can be executed to return a set of Fragments.

2.3.4 Commands

The [editCommands](#) method returns a list of [EditCommand](#) objects listing all specialized commands which operate on Fragments controlled by the expert.

2.3.5 Sentinels

The [sentinels](#) method returns a list of [Sentinel](#) objects which can perform arbitrary semantic analysis and processing for Fragments controlled by the expert.

2.3.6 Importation and Exportation

File importation is controlled by two methods: [filenameExtensions](#) returns a list of strings which are typical file extensions for Fragments controlled by the expert. [importFile](#) takes a filename with one of these extensions and converts it into a list of Fragments.

File exportation is supported by [toObjectFilename](#) and [matchingExports](#). The first function maps from source-file names to object-file names to facilitate our Makefile support. The second one converts an arbitrary keyword to a set of "relevant" exported files in order to support keyword substitution in the "compile" dialog.

2.4 QueryDialog & QueryPanel

The [QueryDialog](#) class is used to create a single dialog which allows the user to access all generally available [Querys](#). It is extended through [LanguageExperts](#), which can each supply an arbitrary number of QueryPanels.

The [QueryPanel](#) class is closely linked to Query. Any user-visible Query will have an associated QueryPanel which collects a set of parameters and then (upon request) creates an instance of the corresponding Query. Any QueryPanel which is registered by a LanguageExpert will automatically be included in the QueryDialog.

2.5 EditCommand

The [EditCommand](#) class provides a universal framework for performing some single operation which may affect the Sheets database. EditCommands may easily be bound to keys in the user [Profile](#), or placed in menus. Most commands are typically registered by an appropriate [LanguageExpert](#).

When EditCommands are invoked, they are passed a target [ViewPanel](#) and a selected [Viewer](#). The command may provide facilities for *checking* its applicability to the targets and *executing* an appropriate action. The framework automatically records any changes for later reversal via the [UndoDialog](#).

Commands may inherit special behaviors by extending the classes [ContainerCommand](#), [ChangeContainerCommand](#), [LethalChangeContainerCommand](#), [CursorMotionCommand](#), or [ChangeContentsCommand](#) or by inheriting from the interfaces [MultiCommand](#), [IterativeMultiCommand](#), [SpecialMultiCommand](#), or [NoArgCommand](#).

2.6 Profile

The [Profile](#) class provides a placeholder for any static variables which can modify the general behavior of Sheets. In addition, it encapsulates facilities for setting the values of these variables and mapping key-bindings based upon the contents of the user's `.sheetsrc`.

Whenever you add a new static field is added to Profile, you should edit [initVariables](#) to provide a default value and [trySet](#) to read the value from `.sheetsrc`. You may wish to make use of the utility functions [getBoolean](#), [getString](#), [getName](#), or [getInteger](#).

2.7 UndoDialog (and support classes)

The [UndoDialog](#) class provides users with a way to undo the results of one or more [ContainerCommands](#) which changed the Sheets database. It uses the [Checkpoints](#) gathered by the [HistoryManager](#) and [HistorySentinel](#) classes to determine what the state of the world was before the command was executed, restores the old state, and then attempts to update [Viewers](#) to reflect the difference. (The former operations are currently fairly solid and robust, but the latter is known to have some problems.)

2.8 Console

The [Console](#) class provides several ways to convey extra information to the user. Most "messages" of any variety are transmitted through static methods on this class.

The most import methods are probably [status](#), which updates the status line within the "context dependent help" window; [display](#), which pops up a message dialog and (usually) waits for confirmation; [beep](#), which lets the user know that he's made a boo-boo; and [debug](#), which just prints to console to assist programmers in finding problems.

3 Semantic Analysis

3.1 ListConstraint

The [ListConstraint](#) class can be used to control precisely what [Fragments](#) are allowed in a [FragmentList](#). It acts directly to prevent addition of *forbidden* fragments or removal of *required* fragments, but relies upon the assistance of some [Sentinel](#) to ensure that required fragments will be added as soon as they are created. Constraints can be created as a result of executing a [Query](#) or at the request of an [Attribute](#) object.

We actually have three categories of fragments: required, permitted, and forbidden. The routines [isRequired](#) and [isForbidden](#) directly define two of these categories. Whatever is left over is permitted.

Architectural Overview

Sometimes it is appropriate for a user to be able to get rid of a constraint that seem to be getting in the way. This can be accomplished by overriding the [canBeRemoved](#) method. (This is done automatically if you inherit from [GenericQueryConstraint](#).)

The trickiest part of developing a constraint comes in ensuring that any required fragments which are newly created end up in the lists. We cannot automate this in the general case (especially for list attributes), so the creator of a constraint is simply held responsible. Typically, he'll assign a Sentinel to do the job.

For most constraints, it is best to use the [AffectedBySentinel](#). This sentinel collects pretty much all the info which might be relevant and passes it on to the [handleChanges](#) routine. All query constraints are automatically added to it's list of constraints. Other constraints may be added via [AffectedBySentinel.addConstraint](#). (We recommend against using it for constrained attributes for efficiency reasons.)

Notes:

- It is easiest to implement a new constraint by extending [GenericListConstraint](#).
- Don't forget to make sure that any newly-created required fragments end up in the right place. [AffectedBySentinel](#) can help.
- Remember that constraints are stored in the database. Extend the [PObject](#) interface for all non-transient fields.
- If you wish to constrain a list-valued attribute, extend [ListAttribute.makeConstraint](#).
- You should probably not use [AffectedBySentinel](#) for constrained attributes.

3.1.1 Query constraints:

There will typically be a ListConstraint class for each [Query](#) class. We have not conflated them because they require different inheritance hierarchies, and because we could conceivably create several constrained lists from a single query.

Extend [Query.createConstraint](#) to associate a constraint with a query. Remember that the constraint is persistent, so be careful what you store in it.

Most queries will return a subclass of [GenericQueryConstraint](#), which already does the work of registering with an [AffectedBySentinel](#).

3.2 Sentinel

The [Sentinel](#) class provides a convenient extension framework for supporting incremental semantic analysis or maintenance of [ListConstraints](#). Sentinels must be *registered* via a

[LanguageExpert](#), and will then receive event notifications whenever something happens to any [Fragment](#). Typically it will be extended to detect creation and deletion of some variety of Fragments, and to update its internal records to reflect the new state of the world. (Note that this notification will occur even if the fragment dies a "natural death" through no longer being referenced anywhere.)

Most Sentinels simply perform semantic analysis for Fragments which are added to or removed from the overall *program*. They can do this by simply responding to [AscensionEvents](#) and [DescentEvents](#) for which [affectsInTheProgram\(\)](#) is true. However, they can receive and can process [ContainerEvents](#) and [FragmentEvents](#).

3.3 ContainerEvent

The [ContainerEvent](#) class is used to notify Observers and [Sentinels](#) when something happens to a [FragmentList](#) which will change the way it should be displayed. In practice, a ContainerEvent is how FragmentLists communicate with the [Viewers](#) for the lists.

ContainerEvents are created when Fragments are [added to](#), [removed from](#), or [moved within](#) a FragmentList. They are also created when the list is [totally changed](#) via a [global undo](#) operation, or if a contained fragment is [replaced](#). (Note that there is a distinct [FragmentEvent](#) which is also triggered for replacements. Both events are always generated, but they are typically delivered to different observers.)

3.4 FragmentEvent

The [FragmentEvent](#) class is used to notify Observers and [Sentinels](#) when something interesting happens to a Fragment. This is primarily for keeping [Viewers](#) in sync with [Fragments](#), although the mechanism is also used by the clipboard and a few Sentinels.

FragmentEvents when Fragments change status by being [replaced](#), [modified](#), changing [export state](#), or being put into *edit mode*. (Note that there is a distinct [ContainerEvent](#) which is also triggered for replacements. Both events are always generated, but they are typically delivered to different observers.)

3.5 AscensionEvent & DescentEvent

The [AscensionEvent](#) and [DescentEvent](#) classes are used to notify [Sentinels](#) of changes to the reachability (*plane of existence*) of a [Fragment](#). The plane of existence can have one of four values: [IN THE PROGRAM](#) indicates that the fragment is reachable from the root (Projects) sheet, and should be considered in global semantic analysis. [HANGING OUT](#) indicates a fragment which is contained in some (presumably temporary) container, but is not reachable from the root sheet. [GONE BUT NOT FORGOTTEN](#) represents fragments which must be retained in the database, but do not appear in any containers. This mostly

Architectural Overview

represents Fragments retained in the global undo history. [WAY TOTALLY DEAD](#) refers to fragments which should be deleted from the database and to which you should retain no pointers.

AscensionEvents are created when Fragments become more reachable, typically by being added to a container during creation, pasting, or global undo. DescentEvents are created when Fragments become less reachable, typically by being replaced or removed from a container. The events contain the Fragment which is affected, and a bit-mask indicating what planes-of-existence are changing. Typically, you will simply call one of: [affectsInTheProgram](#) or [affectsHangingOut](#) to determine how the reachability changed.

4 Persistence

4.1 DB

The [DB](#) class provides a simple object-oriented database which holds [PObjects](#). The interface is encapsulated in a bunch of methods which are static because there can only be one database open at any single time. Allowing multiple open databases might be useful, but then you get into the whole problem of what database is an id relative to and how do you represent cross-database references.

The general idea is that you [open](#) a database, use [State.getGlobalStateVar](#) and [State.setGlobalStateVar](#) to get and set named *roots*, call [commitChanges](#) whenever you want to checkpoint your state, and then finally call [close](#) when you are done.

Any PObject that gets stored in the database is assigned an integer ID via [assignID](#). Anyone who wants to reference some database object records the ID for that object. When they need the actual object, they use [lookupID](#) to map the ID to an actual object. Object references are indirected through an explicit ID lookup so that having one object in memory does not require that all of the objects it references also be loaded into memory.

There are two magical IDs which you may wish to make use of -- [nullID](#) represents the "null" object, while [deletedID](#) can be used to warn the system that a reference is dead.

When a PObject is no longer used, it should be explicitly removed via [freeID](#). There is no automatic garbage collection, so items which are not freed will simply accumulate and eat up disk space. Typically PObjects will create a "free" method which not only calls freeID for its own ID, but also cleans up any other PObjects which it owns.

When some detail about a object stored in the database changes, [noteIsDirty](#) must be called on the object so that the database knows to schedule a writeback of the object. The typical way this is handled is to have all fields be private to the class and offer [getMumble](#) and [setMumble](#) methods. The [setMumble](#) method just calls [DB.noteIsDirty](#) if the field actually

changes. The `get/setMumble` methods can also automatically convert object IDs into objects making it look like the field holds the actual object when in reality it just holds the ID.

DB provides a few extra utility routines: [readInt](#) and [writeInt](#) provide compact and fast storage of any length of integer. [validate](#) and [printUsageSummary](#) are used primarily for debugging detecting leaks. The former checks the consistency of the database, while the latter determines how much space is being used up by each variety of PObject.

4.2 PObject

All persistent objects must implement the [PObject](#) interface and either implement the [Serializable](#) interface or be *registered* by a [LanguageExpert](#). The interface is described in detail below. For information on the [Serializable](#) and [Externalizable](#) interfaces, see the Java [Serialization](#) documentation.

4.2.1 The Interface

The PObject interface consists of four methods. [loadData](#) and [storeData](#) must do explicit reads and writes for all of the persistent data, calling `super` as necessary. (If the object is [Serializable](#) and not registered, these won't be called, but must still have trivial definitions.) [getID](#) returns the ID for the object, calling `DB.assignID` if necessary. [free](#) recursively calls `free` on any sub-objects if necessary and then calls `DB.freeID` to release the DB storage for the object. If Java allowed interfaces to supply default implementations, your life would be a lot simpler, but well, it doesn't.

Something along the lines of:

```
private int id = DB.nullID;

public int getID ()
{
    if (id == DB.nullID)
        id = DB.assignID(this);
    return id;
}

public void free ()
{
```

```
// recursively call free if necessary.  
if (id != DB.nullID)  
    DB.freeID(id);  
id = DB.deletedID;  
}
```

should do the trick.

4.2.2 Fields in Persistent Objects

Most PObjects implement `readObject` and `writeObject` routines which explicitly dump each of the simple fields. If they are arrays or Vectors, you'll have to explicitly read and write elements. This is tedious, but immensely faster and more compact than the serialization protocol. You **must** make each of these classes public and implement a null constructor (i.e. one with zero arguments which doesn't bother initializing the fields which will be loaded).

If you must use the serialization protocol, you should make the class `Serializable`, and should **not** register it with any `LanguageExpert`. You must make sure that all non-transient fields are themselves `Serializable`. All of the primitive types and many others, like `Vector` and `String`, are serializable, so this isn't a big restriction. If you flag a field as transient, the serialization stuff won't save or restore it. You can use this to store cached values in memory without forcing that value to be stored to disk. When the object is read back in, the transient field will end up as the default value for that type (NOT the initial value, if one was supplied). You can also use transient fields along with `readObject` and `writeObject` methods to save the field in a different format than the serialization stuff would itself.

Absolutely never store one PObject directly inside another PObject. This won't work because when the output PObject gets written, the inside PObject will be written right along with it. Then when the outside PObject gets loaded back in, a duplicate of the inside PObject will be created. This would result in multiple distinct objects existing in memory for the same ID. Instead, store the ID of the PObject in an int field, and call [DB.lookupID](#) whenever you need the object. If you absolutely don't want to call `DB.lookupID`, then use a transient field to cache the results of the lookup. But `DB.lookupID` is fast enough that you shouldn't have to bother.

4.2.3 Changing fields

If a field ever changes, you need to call [DB.noteIsDirty](#) on the PObject. This queues the object to be written back to the database. You don't need to immediately call `noteIsDirty`, you just have to guarantee that it will be called. So you can make multiple changes and then call `noteIsDirty` once, if you want. But calling `noteIsDirty` should be sufficiently fast that you

shouldn't have to. The simplest solution is to just offer `setMumble` methods for all your mutable fields and have them call `noteIsDirty`.

The one exception is that you don't need to call `noteIsDirty` if `getID` has never been called. This is because objects are only stored into the database if someone asks for the ID for the object. In fact, calling `noteIsDirty` will cause `getID` to be called, so calling `noteIsDirty` will force the object to be stored into the database even if nobody else is actually referencing it.

So if you can guarantee that nobody will have asked for the object ID yet, you don't need to call `noteIsDirty`. One place this routinely happens is in constructors. Until the constructor finishes, nobody outside the constructor can have their hands on the object to be able to call `getID`. So unless some super constructor method calls `getID` (or causes `getID` to be called) you can skip the call to `noteIsDirty`.

A common idiom is to offer a protected `stateChanged` method that checks to see if an id has been assigned, and only calls `noteIsDirty` if one has. For example:

```
protected void stateChanged () {  
    if (id != DB.nullID)  
        DB.noteIsDirty(this);  
}
```

Then your `setMumble` methods can call `stateChanged` without worrying about whether or not doing so will cause garbage to accumulate in the database.

4.2.4 Identity

The database stuff carefully preserves identity for PObjects. What this means is that two distinct PObjects will never have the same ID and two different lookups of the same ID will always result in the same (`==`) PObject. Well, at least as far as you can tell, that is. If you lookup some ID and then drop all references to the resultant PObject, the database is free to let the system garbage collect that PObject and return you a new one the next time that ID is looked up. But if you keep that original PObject live, the database will always return it for successive lookups.

So you can just use `==` to check to see if two PObjects are in fact the same PObject.

This also means that you can meaningfully make Hashtables that use PObjects as keys. These hashtables can't be stored in the database because they directly reference PObjects, but they can be used ephemerally during a single session.

4.2.5 The ProtoPObject class.

Architectural Overview

The [ProtoPObject](#) class implements the PObject interfaces, supplies implementations for all the interface methods, and also offers a stateChanged method as described above. (You will, of course still have to implement readObject and writeObject methods to handle all directly defined fields.)

If your persistent object doesn't need to extend something else (Dictionary, for example), you can just extend ProtoPObject and pick up those methods for free. Then all you have to do is make sure you follow the conventions for fields as described above.

The [SerializablePObject](#) class implements the above interfaces and also implements Serializable. If you extend this, you won't have to implement readObject and writeObject methods and shouldn't register the class with any LanguageExpert. On the other hand, this will be apallingly slow and wasteful of space.

4.3 PVector

The [PVector](#) class is similar to Vector, except that it stores [PObjects](#) and is a PObject itself. All of the operations available on Vectors (except the capacity stuff) are available on PVectors. In addition, there is a [toVector](#) method for converting a PVector into a Vector, and a [free](#) operation to remove it from the [DB](#) when the PVector is no longer needed.

4.4 PTable

A [PTable](#) is a persistent Dictionary that can map [PObject](#) or Serializable keys to PObject or Serializable elements. It uses a linear hashing mechanism to keep the cost of rehashing down and uses large buckets to keep the number of disk accesses per lookup low.

Note: PObject keys will use "getID" instead of "hashCode". The reasons are obscure -- primarily because we need to use an ephemeral hashcode in order to compute the persistent ID.

4.5 PObservable & PObserver

The [PObservable](#) class is similar to Observable, except that it correctly handles persistent (i.e. [PObject](#)) observers. At present, all the bogosities of the Observable interface are faithfully reproduced in the PObservable interface, but that might change.

Note that PObservable is not a PObject but does implement the [loadData](#) and [storeData](#) methods. This means that by default, subclasses of PObservable can call super.loadData and super.storeData to ensure that the state of the PObservable will be saved.

[PObservers](#) do not have to be PObjects either. If a PObserver is a PObject, it will correctly be handled and the observing relationship will be saved in the database. But if the PObserver is

not a POject, the observing relationship will last only as long as the PObservable is kept in memory. So any non-POject PObservers should keep ahold of the PObservable itself (not just its ID) to make sure it doesn't get collected and read back in later.

PObservable implements a few utility methods that might be useful for various users. The [setChangedAndNotify](#) routine allows any class to make sure notification happen (unlike notifyObservers which relies upon the protected setChanged method). The [readStrings](#), [writeStrings](#), and [writeStringsOrNull](#) methods are simple routines for reading and writing arrays of strings. (They don't really belong in this class, but they don't fit anywhere else either.)

4.6 ArchivalObject

The [ArchivalObject](#) class represents objects which are persistent and can be read and written to a *dump file* using a [FragmentReader](#) and [FragmentWriter](#). (At this point, this just consists of [Fragments](#), but there could be others.) The ArchivalObject holds the state to be saved, in is subclassed by various kinds of fragments in the database. The split between ArchivalObject and FragmentReader/Writer allows fragment classes to be added without awareness of all the ways they might be archived, and similarly new kinds of archiving to be added without awareness of what all the fragment classes are.

We inherit from [PObservable](#) because Java is single-inheritance and Fragment must inherit PObservable.

Subclasses must implement [saveCoreInfo](#) and add to the [typesTable](#) a [LanguageExpert](#) which can load the fragment again. They must also implement an [isDumped](#) method which computes whether this particular object should be written out at all.

Subclasses can also optionally override the default do-nothing methods [saveAuxiliaryInfo](#), [loadAuxiliaryInfo](#), and [finalizeLoad](#). These methods must do their saving and loading by using the interfaces offered by fragment reader/writer.

All archival fragments must either have an objectId or be a [LightWeightFragment](#). You can assign an archival fragment an id by calling [useIdFrom](#) or [useUniqueId](#).

4.7 FragmentReader & FragmentWriter

The [FragmentReader](#) and [FragmentWriter](#) classes provide an abstract interface for reading and writing the non-derived state of an [ArchivalObject](#), assisted by an appropriate [LanguageExpert](#). (This process is sometimes referred to as *dumping* and *undumping*.) This is used to provide a compact encoding suitable for data interchange and database evolution.

The only readers and writers currently supported are [AsciiFragmentReader](#) and

Architectural Overview

[ntWriter](#), which provide a textual representation suitable for human eyes and typical revision-control systems.

Sheets automatically uses the readers and writers to read and write interchange or version databases. There should be little need for others to call these routines. However, each subclass of [ArchivalObject](#) must be extended to support the process. For writing, they should extend [SaveCoreInfo](#) and possibly [saveAuxiliaryInfo](#). For output, they depend upon a [LanguageExpert](#) to provide an appropriate [loadFragment](#) routine, but may also wish to extend [loadAuxiliaryInfo](#) or [finalizeLoad](#).

4.8 LightweightFragment

The [LightWeightFragment](#) class represents [ArchivalObjects](#) which are considered equivalent to all other [ArchivalObjects](#) of the same class. For instance, all separators are basically the same, so a [loadSeparator](#) method would just cons up a new one rather than try to recreate the old one. For that reason, we don't really need to save light weight fragments to disk.

This interface has no real semantics associated with it; it's just useful for doing instance of checks. This interface is not a true subclass of [ArchivalObject](#) because Java doesn't let interfaces extend classes.

The only thing you have to do is write a [getObjectId](#) method that returns "lightWeight:whatever" and modify [OIS.makeLightWeight](#) to understand it.

5 Java Support

5.1 JavaFragment

Instances of the [JavaFragment](#) class represent the various pieces of code which make up a Java program. Various subclasses represent the various different types of code: [JavaFragment](#), [JavaFileHeaderFragment](#), [JavaClassFragment](#), [JavaInitializerFragment](#), [JavaConstructorFragment](#), [JavaMethodFragment](#), [JavaFieldFragment](#), [JavaCommentFragment](#), [UnknownJavaFragment](#).

New instances of [JavaFragment](#) are created by [JavaExpert](#), aided by [JavaFragmentParser](#). They are viewed through the [JavaViewer](#) class, and inter-fragment semantic relationships are tracked through the [JavaSentinel](#) and [JavaPackageSentinel](#). The meaning of the fragments text is interpreted through the [JavaTokens](#) class, which breaks it up into tokens and then uses ad-hoc techniques to determine the meanings of those tokens.

5.2 JavaViewer

The [JavaViewer](#) class is responsible for displaying all varieties of [JavaFragments](#). Most of its

work is done by simply deferring to [JavaTokens](#) for syntax highlighting, indentation, identifier completion, and context-sensitive help.

5.3 JavaTokens

The [JavaTokens](#) class accepts a piece of text which is presumed to correspond to Java code and maintains a corresponding set of Tokens. These tokens are used by [JavaViewers](#) to perform syntax highlighting. They are also analyzed in an ad-hoc manner to attempt to determine the meaning of the code from these tokens. This information is used to assist with context-sensitive help, identifier completion, and Java-related [queries](#).

It attempts to be as robust as possible, since the code is expected to be applied to code which is in the process of being edited. The modification is facilitated by the two methods [replaceLines](#) and [replaceMultiLines](#).

5.4 JavaSentinel & JavaPackageSentinel

The [JavaSentinel](#) and [JavaPackageSentinel](#) classes work together to gather and store information about all of the [JavaFragments](#) in the system. This is used to track the inter-relationships between fragments, and to aid in Java related [Queries](#).