

JEDIT

SOURCE CODE INVESTIGATION

Assignment 2



Group: Friday 3pm C

Members: Timothy Brown - 40064026

Kathleen Finch - 40083269

Andrea McAully - 40112888

Timothy Woollams - 40341129

Table of Contents

Introduction.....	4
Overview of Structure.....	4
Package Descriptions	4
List of Classes	5
External Libraries.....	12
Detailed Description	12
Class Subset Justification.....	12
Class Analysis.....	12
jEdit.....	12
EditBus.....	13
EBMessage	14
EditPlugin.....	14
JAR.....	15
Broken	16
EBPlugin	16
EditAction	16
ActionSet.....	17
Buffer	18
BufferHistory	19
Marker	20
Mode	21
Registers.....	21
TextUtilites	22
MiscUtilities.....	23
UML Diagram.....	24
Patterns.....	25
CVS and Unix Build Evidence	<i>not supplied</i>
Team Learning Journal	<i>not supplied</i>
Plan for Future Research.....	<i>not supplied</i>
Extension of the Design Documentation	<i>not supplied</i>
Possible Improvements, Extensions and Refactoring.....	<i>not supplied</i>
Testing.....	<i>not supplied</i>
Conclusion	<i>not supplied</i>
References	39

Introduction

JEdit is a text-editing program. It is an open source program written in Java. Programmers are free to add and modify the source code and plugins. The submission must be approved however, and the approved code can be added to new versions. JEdit is a programmers text editor. While it can be used for editing plain text documents it does not include features for text formatting such as bold, italics, etc. that are found in other word processor orientated editors. However, it is very useful for editing code. Almost all of its features and plugins are specifically designed for this type of editing. It's true capabilities can only be found when it is used in conjunction with a number of plugins. For example, plugins can be added which allow source code to be compiled within JEdit.

Some of the features include

- Syntax highlighting
- Automatic indenting of source code
- Built-in macro language
- Folding
- Word-wrap
- Unlimited undo/redo
- Extensible plugin architecture
- Character encoding
- Search and replace
- File management

There was not much existing documentation that we could find to assist with the creation of this document. There was a help file in jEdit, which provided a small amount of assistance. Most of the information below was gathered by reading through the source code and slowly trying to determine its' meaning.

Overview of Structure

The source code for jEdit is divided into several packages. These packages represent general groupings of the different functions that are available in jEdit. There are brief descriptions of each package below. Each package contains a varying number of classes, which are all listed below. We will also describe some of the more important classes in detail.

Packages Descriptions

bsh

This is the beanshell package. It includes all the classes related to interpreting beanshell scripts.

bsh.ReflectManager

This only has one class, reflectManagerImpl.

gnu.regexp

It contains classes including and relating to the class RE (regular expression).

installer

The package used for installing jEdit.

org.gjt.sp.jedit

This package contains jEdit's core classes. Some of them are not as vital to the running of jEdit as others, but they fit most appropriately in this package rather than any other.

org.gjt.sp.jedit.browser

The package of classes making up jEdit's file system browser.

org.gjt.sp.jedit.buffer

Buffer event listeners, and classes internal to jEdit's document model. It also has the classes needed for using the fold functionality.

org.gjt.sp.jedit.gui

This package has the classes that are responsible for the appearance of jEdit. It includes various GUI controls and dialog boxes.

org.gjt.sp.jedit.io

This packages deals with the input and output aspects of jEdit, including Virtual File Systems and multi-threaded I/O.

org.gjt.sp.jedit.msg

This contains the classes responsible for the EditBus messages.

org.gjt.sp.jedit.options

The package has the GUI classes for options dialog boxes.

org.gjt.sp.jedit.pluginmgr

This package contains the plugin manager classes. These look after the plugins that are installed in jEdit.

org.gjt.sp.jedit.print

The package dealing with printing. Its classes include attributes that store the information regarding the layout of the text.

org.gjt.sp.jedit.search

Contains all the classes needed for the search and replace function.

org.gjt.sp.jedit.syntax

Includes the classes responsible for performing the syntax highlighting of source code.

org.gjt.sp.jedit.textarea

This is jEdit's text area control. Its main class is jEditTextArea, which has all the methods relating to the visual component of editing text.

org.gjt.sp.util

This appears to contain miscellaneous utility classes used by jEdit.

List of Classes

The following is a complete list of classes divided into the packages that they belong to.

Package: bsh

ASCII_UCodeESC_CharStream.java

BlockNameSpace

BSHAllocationExpression
BSHAmbiguousName
BSHArguments
BSHArrayDimensions
BSHArrayInitialiser
BSHAssignment
BSHBinaryExpression
BSHBlock
BSHCastExpression
BSHClassManager
BSHFormalComment
BSHFormalParameter
BSHFormalParameters
BSHForStatement
BSHIfStatement
BSHBSHImportDeclaration
BSHLHSPrimaryexpression
BSHLHSPrimarySuffix
BSHLiteral
BshMethod
BSHMethodDeclaration
BSHMethodInvocation
BSHPrimaryExpression
BSHPrimarySuffix
BSHPrimitiveType
BSHReturnStatement
BSHReturnType
BSHStatementExpressionList
BSHSwitchLabel
BSHSwitchStatement
BSHTernaryExpression
BSHThrowStatement
BSHTryStatement
BSHType
BSHTypeVariableDeclaration
BSHUnaryExpression
BSHVariableDeclarator
BSHWhileStatement
CallStack
Capabilities
ClassPathException
CommandLineReader
ConsoleInterface
EvalError
Interpreter
InterpreterError
JJTParserState
JThis
LHS
Name
NameSource
NameSpace
Node
ParseException
Parser

ParserConstants
ParserTokenManager
ParserTreeConstants
Primitive
Reflect
ReflectError
ReflectManager
ReturnControl
SimpleNode
StingUtil
TargetError
This
Token
TokenMgrError
XThis

Package: bsh.ReflectManager

ReflectManagerImpl

Package: gnu.regexp

CharIndexed
CharIndexedCharArray
CharIndexedInputStream
CharIndexedReader
CharIndexedString
CharIndexedStringBuffer
RE
REException
REFilterInputStream
REFilterReader
REMatch
REMatchEnumeration
RESyntax
REToken
RETokenAny
RETokenBackRef
RETokenChar
RETokenEnd
RETokenEndSub
RETokenLookAhead
RETokenOneOf
RETokenPosix
RETokenRange
RETokenRepeated
RETokenStart
RETokenWordBoundary
UncheckedRE

Package: installer

ConsoleInstall
ConsoleProgress
Install
InstallThread
JEditMetalTheme
NonInteractiveInstall

OperatingSystem
Progress
SwingInstall

Package: org.gjt.sp.jedit

Abbrevs
AbstractOptionPane
ActionListHandler
ActionSet
Autosave
BeanShell
BeanShellAction
Buffer
BufferHistory
EBComponent
EBMessage
EBPlugin
EditAction
EditBus
EditPane
EditPlugin
EditServer
GUIUtilities
JARClassLoader
Java14
jEdit
Macros
Marker
MiscUtilites
Mode
ModeCatalogHandler
OperatingSystem
OptionGroup
OptionPane
Registers
SettingsReloader
TextUtilities
View

Package: org.gjt.sp.jedit.browser

BrowserCommandsMenu
BrowserIORequest
BrowserListener
BrowserView
FileCellRenderrer
VFSBrowser
VFSFileChooserDialog

Package: org.gjt.sp.jedit.buffer

BufferChangeAdapter
BufferChangeListener
ContentManager
DummyFoldHandler
FoldHandler

IndentFoldHandler
LineElement
OffsetManager
RootElement
UndoManager

Package: org.gjt.sp.jedit.gui

AbbrevEditor
AboutDialog
AddAbbrevDialog
BeanSchellErrorDialog
BufferOptions
BufferSwitcher
CloseDialog
ColorWellButton
CompleteWord
DefaultInputHandler
DirectoryMenu
DockableWindowManager
EditAbbrevDialog
EnhancedButton
EnhancedCheckBoxMenuItem
EnhancedDialog
EnhancedMenu
EnhancedMenuItem
ErrorListCellRenderer
ErrorListDialog
FloatingWindowContainer
FontSelector
GrapKeyDialog
HelpVeiwier
HHistoryModel
HistoryTextField
InputHandler
IOProgressMonitor
JCheckBoxList
JEditMetalTheme
KeyEventWorkAround
LogVeiwier
MarcrosMenu
MarkersMenu
OptionsDialog
PanelWindowContaner
PastePrevious
RecentDirectoriesMenu
RecentFilesMenu
RolloverButton
SelectLineRange
SplashScreen
StausBar
TipOfTheDay
ToolBarManager
VariableGridLayout
ViewRegisters

Package: org.gjt.sp.jedit.io

BufferIORequest
FavouritesVFS
FileRootsVFS
UrlVFS
VFS
VFSManager

Package: org.gjt.sp.jedit.msg

BufferUpdate
EditorExiting
EditorExitRequested
EditorStarted
EditPaneUpdate
MacrosChanged
PropertiesChaged
SearchSettingsChanged
VFSUpdate
ViewUpdate

Package: org.gjt.sp.jedit.options

AbbrevsOptionPane
AppearanceOptionPane
BrowserColorsOptionPane
ColorOptionPane
ContextOptionPane
DockingOptionPane
EditingOptionPane
FirewallOptionPane
GeneralOptionPane
GutterOptionPane
LoadSaveOptionPane
MOModeOptionPane
OverviewOptionPane
PrintOptionPane
ShortcutsOptionPane
StyleOptionPane
TextAreaOptionPane
ToolBarOptionPane

Package: org.gjt.sp.jedit.pluginmgr

InstallPluginsDialog
PluginList
PluginListDownloadProgress
PluginListHandler
PluginManager
PluginManagerProgress
Roster

Package: org.gjt.sp.jedit.print

BufferPrintable
BufferPrinter1_3
BufferPrinter1_4

Package: org.gjt.sp.jedit.search

AllBufferSet
BoyerMooreSearchMatcher
BufferListSet
CharIndexedSegment
CurrentBufferSet
DirectoryListSet
HyperSearchRequest
HyperSearchResult
HyperSearchResults
RESearchMatcher
SearchAndReplace
SearchBar
SearchDialog
SearchFileSet
SearchMatcher

Package: org.gjt.sp.jedit.syntax

Chunk
DefaultTokenHandler
DisplayTokenHandler
DummyTokenHandler
KeywordMap
ParserRule
ParserRuleSet
SoftWrapTokenHandler
SyntaxStyle
Token
TokenHandler
TokenMarker
XModeHandler

Package: org.gjt.sp.jedit.textarea

ChunkCache
ExtensionManager
FoldVisibilityManager
Gutter
JEditTextArea
ScrollListener
Selection
TextAreaExtension
TextAreaPainter

Package: org.gjt.sp.util

IntegerArray
Log
ReadWriteLock
SegmentBuffer
WorkRequest
WorkThread
WorkThreadPool
WorkThreadProgressListener

External Libraries

All the required classes were contained within the jEdit source code archive except for the standard classes that belong to the basic java system.

Detailed Description

Class Subset Justification

As is evident from the list of classes above there is much code that we could chose to analyse. We decided to investigate the some of the classes surrounding the main class, jEdit.java. We tried to focus of the classes that were the basis of the running of the program. However, the functionality of jEdit is obviously spread throughout the whole source code. Therefore we included some classes in our analysis that initially seemed important but under closer scrutiny showed that they did not assist greatly with the core running of jEdit.

Class Analysis

All attributes and methods mentioned below are private and public respectively unless otherwise stated. Likewise, if a return type has not been mentioned for a method, that method does not return a value.

It should be noted that all deprecated methods (i.e. those that are no longer recommended and will not be supported in later versions) have been excluded.

JEDIT

This is the main class of the jEdit text editor. It is a large very class and includes many methods. There is not time to analyse all of them, so a brief description of a few of the focal methods are written below.

Attributes

Name	Type	Description
JEditHome	String	The jEdit install directory
actionSets	Vector	The set of ActionSets contained in the actions.xml file
pluginErrors	Vector	A list of error entries concerning the plug-ins
jars	Vector	A vector of objects of the class JAR e.g. jar files for plug-ins
modes	Vector	A list of all edit modes contained within the catalogues
inputHandler	InputHandler	An object that converts keystrokes into concrete actions.
props	Properties	A HashTable that stores a collection of properties of the program

Methods

openFile(View view, String path) – opens the file, named in the path, in the specified view.

It returns a *Buffer*.

newFile(View view) – This returns a *Buffer* in the specified view, which is given the caption “Untitled <num>”.

closeBuffer(View view, Buffer buffer) – Closes the given buffer in the given view. It returns a *Boolean*, which is true if the close is successful. If there are unsaved changes the user will be prompted to save the buffer.

saveAllBuffers(View view, Boolean confirmed) – This saves all unsaved, open buffers in the view. If confirmed is true then jEdit will ask for a confirmation of the action.

closeAllBuffers(View view) – Closes all open buffers in the specified view and prompts for the user to save each buffer with unsaved changes in it. It returns a *Boolean* which is true if the close was successful.

exit(View view, Boolean reallyExit) – this closes jEdit, but if it is running in background mode and reallyExit is false it will close all buffers and views and remain in background mode.

getProperty(String name) – returns a *String* which is the value of the name property.

setProperty(String name, String property) – sets the property with that name to the given value. Existing properties are overwritten.

getBooleanProperty(String name) – returns the *Boolean* value of the property.

setBooleanProperty(String name, Boolean property) – sets the property with that name to the given Boolean value. Existing properties are overwritten.

getIntegerProperty(String name) – returns an *int* value for the property and if the property specified is not a valid numeric string it returns the default value instead.

setIntegerValue(String name, int value) – sets the property given to the value given.

getActions() – returns an *EditAction* array, that are the actions maintained and used by the editor.

getAction(String action) – returns an *EditAction* that is named as a parameter.

getBuffers() – returns a *Buffer* array, which includes all the currently open buffers.

getBuffer(String path) – returns the *Buffer* containing the file given in the path.

getMode(String name) – returns the *mode* specified.

getPlugin(String name) – returns an *EditPlugin* that has that name.

The following methods are all private

initMisc() – this method does a few things. It creates an input handler, a JAR vector, adds to the editBus the component that reloads editModes and macros as well other things.

initSystemProperties() – creates new Properties and loads the properties.

initActions() – creates an actionSet vector, creates the built-in actionSet and adds this to the actionSet vector. Also loads XML actions.

initPlugins() – loads the plugins using the next method.

loadPlugins(String directory) – finds the directory and any plugins in it are stored.

EDITBUS

This class acts as the central mechanism of communication between the components of jEdit. This includes plugins, buffers and other non-trivial classes. It distributes messages described within the org.gjt.sp.jedit.msg package between all components, functioning much like a hardware bus. The entire class is static and therefore only one EditBus exists in the jEdit program demonstrating the Singleton pattern. The constructor does not allow new instances of EditBus to be created.

All components on the EditBus extend the EBComponent class. Most EBComponents must add themselves to the EditBus during initialisation however plugins will be automatically added during the starting up of the program in the jEdit class.

Attributes

All of the attributes of EditBus are static.

Name	Type	Description
components	ArrayList	An ArrayList of EBComponents to keep track of which components will send and receive messages through the bus
copyComponents	EBComponents[]	The components are also stored in this array to return during the getComponents() method

Methods

addToBus(EBComponent comp) – This method adds another component to the bus' list.

The component will now receive all global messages from within the system.

removeFromBus(EBComponent comp) – This method removes the specified component from the bus' list. The component will now cease to receive all global messages from within the system.

getComponents() – Returns an *array of components* all of which are connected to the bus. It uses copyComponents to do this.

send(EBMessage message) – This method is responsible for distributing a message received to all the components on the bus. To do this it makes a log entry and then creates a temporary array of the components to avoid problems that could arise if EBComponents were added or deleted to the list during the operation. This array is iterated through to send the message. Each EBComponent has a handleMessage() method, which determines how it will deal with each particular message. Any object, not just EBComponents, can invoke this method.

EBMESSAGE <<ABSTRACT>>

This is the class from which all messages that can be distributed throughout the system are created. All messages, which are located in the org.gjt.sp.jedit.msg package, must extend this class.

Attributes

Name	Type	Description
source	EBComponent	EBComponent defines what is responsible for creating the message and sending it to the EditBus

Methods

All of the methods for a particular EBMessage are public non-static methods. They include a getter method for the source and methods relating to displaying a string representation of a message.

EDITPLUGIN <<ABSTRACT>>

All plugins must implement this class as it allows interaction between jEdit and the plugin. If there is a need for the plugin to send and receive global message through the EditBus, the plugin must extend the EBPlugin class instead, since EBPlugin already implements both EditPlugin and EBComponent.

Attributes

Name	Type	Description
jar	EditPlugin.JAR	Each plugin stores its relevant properties in a JAR file. This information is needed to be stored in order for deleting the plugin and retrieving the properties during the initialisation by the jEdit class.

Methods

Apart from the standard getter methods for jar and the plugin class name, EditPlugin contains four additional methods. Although it is not compulsory for these methods to be implemented it is likely that as least some will be required to be implemented for full functionality.

start() – This method is called by jEdit during start up for all installed plugins. It could be used to connect to the EditBus and also contain other instructions to initialise the plugin itself.

stop() – This method is rarely implemented. It is included as a precaution to stop any activities involving the plugin or perform closing operations before the program is terminated.

createMenuItems(Vector menuItem) – This method is called each time a view is created. At this time this method is invoked for each plugin in order to obtain the information that will be contained in the view's Plugin menu. It is loaded using GUIUtilities. The parameter menuItem keeps the list of menus and menu items as it is passed between the plugins.

createOptionPane(OptionsDialog optionsDialog) – This method is called when the global options box for plugins is created. This method is called for each plugin at this time and each plugin adds an optionPane (or group of optionPanes) to the optionsDialog.

JAR

JAR is a static class contained within EditPlugin. Each instantiation of the class is a .jar file, which is an archive of the plugins successfully loaded, their properties and actions.

Attributes

Name	Type	Description
path	String	Contains the path to the relevant .jar file, which keeps track of the plugins successfully loaded so that their routines may be utilized by jEdit. It is a plugin archive.
classLoader	JARClassLoader	An instantiation of the JARClassLoader class which is responsible for reading the .jar archive file and retrieving the plugins in the file
plugins	Vector	The list of plugins that exist in the .jar file
actions	ActionSet	A set of actions associated with the plugins

Methods

The class contains the standard getter methods to retrieve the path, classLoader and actions. It also contains a getter method for the list of plugins, however the plugins are returned as an array instead of a Vector.

Constructor – The class contains one constructor, which requires the path of the file and JARClassLoader as input to set these private variables. The other variables are initialised from this.

addPlugin(EditPlugin plugin) – This method is used to add a plugin to the list of plugins associated with the JAR file. It sets the jar attribute of the EditPlugin passed to be this JAR file and then if the plugin extends the EBPlugin class (i.e. is able to receive messages from the EditBus) it adds the plugin to the Edit bus. Finally the passed EditPlugin is added to the list of plugin associated with this JAR file.

getPlugins (Vector vector) – Takes a vector as a parameter and fills this parameter with the contents of the plugin vector associated with this JAR object i.e. all of the plugins that are archived in this JAR file. It is package-private method.

BROKEN (STATIC)

This is a static class contained within EditPlugin that extends EditPlugin. This is a particular type of EditPlugin. It is used as a holder for any plugins that do not load successfully.

EBPLUGIN <<ABSTRACT>>

EBPlugin extends EditPlugin and implements EBComponent. This class is the template for all plugins as it gives them the structure a plugin must exhibit while allowing it to be connected to the EditBus and therefore receive messages from other components in the system.

Methods

handleMessage(EBMessage message) – this is the method that states how received messages are dealt with. By default the plugin will ignore messages, however each individual plugin may and probably should overwrite this method.

EDITACTION <<ABSTRACT>>

EditAction is the basis from which all actions are specified. This class is used as an action listener for each action and defines GUI related operations for EditActions. It contains a class wrapper, which implements the interface ActionListener. There is a wrapper class associated with each action.

However, jEdit is designed so that the actions do not actually extend this class. Actions define procedures that can be bound to items such as menus or invoked when a particular event occurs. They can perform many tasks, such as causing the appearance and disappearance of plugins (retrieved from jEdit help).

Instead, all actions are created in the action.xml file, which specifies their name, properties and operations they will perform when invoked, written in BeanShell scripting code. Each action must be defined with a name, and the optional boolean properties of:

- no_repeat: indicates whether the user can repeat the action using the short cut Control-Enter.
- no_record: specifies whether the action can be recorded while the user is recording a macro.
- is_selected: that determines the state of the checkbox item associated with the plugin

According to the jEdit help system, “an internal table of this BeanShell scripts means that an ActionListener does not need to be maintained to react to events that occur in the system, even though the EditAction abstract class has allowed for this”.

This .xml file may be included in the jar file associated with the plugins.

Attributes

Name	Type	Description
name	String	This is the internal name of the action. This will be the identifier and string representation of an action

Methods

In addition to a getter method of the private name variable of the action, there is also getLabel() method which returns the label, as a String, that is associated with the action. By default it returns the “value of the property named by the action's internal name suffixed with .label”.

Constructor – Requires the name of the action as the parameter to set the private instance variable.

getMouseOverText() – Returns a copy of the text as a String that should appear when the mouse is over a menu item as a description. This is by default set to null.

<<abstract>> invoke(View view) – This abstract method is used to invoke the code associated with the action. This would be implemented in extensions of the class, however as this class is not extended this method will never be implemented.

<<static>> getView(Component comp) – This method returns a *View*, which is responsible for GUI issues associated with plugins such as creating menu items and managing their DockableWindows. This is a method that has been deprecated in the current class and should technically belong to the GUIUtilities Class. It invokes the getView(Component comp) method in that class. It is passed a Component, which should be a specialisation of JComponent class.

isToggle() – Returns a *Boolean* stating if the EditAction should appear as a checkbox in menus. This is by default set to false.

isSelected (View view) – This returns a *Boolean* if it is a checkbox in menus and is selected. Otherwise it is set to false.

noRepeat() – Returns true if this action may be repeated using the short cut Control-Enter. This is by default set to false.

noRecord() – Returns a Boolean that is true if this action may be recorded. This is set to false by default.

<<abstract>> getCode() – This abstract method when implemented would return the BeanShell code that performs the action in the form of a *String*.

ACTIONSET

An instantiation of ActionSet holds a collection of EditActions.

Attributes

Name	Type	Description
Label	String	This is the identifier and string representation of the set of EditActions
actions	HashTable	All EditActions are stored in a HashTable, using the name of the action as the key

Methods

The class contains a getter method for the label of the ActionSet and a setter method for the label, which takes a String

Constructors – There are two constructors. The default constructor creates a null set. The second constructor takes as a parameter the label of another ActionSet and set the new ActionSet to be equivalent to the one passed.

addAction(EditAction action) – Puts ‘action’ into the Hashtable ‘actions’, using the name of the action as the key.

removeAction(EditAction action) – Removes ‘action’ from the HashTable ‘actions’, using the name of the action as the key to locate it.

removeAllActions() – Removes all actions from the set by clearing the HashTable ‘actions’.

getAction(String name) – Returns the *EditAction* specified by the parameter. The action is located in the HashTable ‘actions’, using the name of the action as the key.

getActionCount() – Returns the *int*, which is the number of actions in the set. This is the size of the HashTable.

getActions() – Returns an *array of EditActions* which are all the ones that exist in the set.

contains(EditAction action) – Returns true if the action specified as a parameter belongs to the set.

getActions (Vector vector) – This method adds every EditAction in the set to the vector passed as a parameter. This is a package-private method.

BUFFER

An object of the Buffer class is “the contents of an open text file as it is maintained in the computer's memory (as opposed to how it may be stored on a disk)”. Buffer.java is 3642 lines long, including comments and blank lines. It contains many methods and is far more extensive than the very brief overview given here, however hopefully the reader will be able to get at least an idea of the sort of methods provided by the Buffer class.

Attributes

Name	Type	Description
flags	int	Holds certain information like if new file, dirty and read only
file	File	Holds a reference to the actual file being represented by the buffer
path	String	Holds the path of the actual file
vfs	VFS	Virtual file system, to provide a more powerful method of accessing the file
properties	Hashtable	Contains properties about the current buffer, such as tab size, indent size and maximum line length
markers	Vector	Markers act as bookmarks to allow the use to easily jump to other parts of the buffer. This structure holds the markers for

		this buffer
mode	Mode	Edit modes define specific settings for editing some types of files. This specifies the edit mode for this particular file.

Methods

These are some I/O methods

load(final View view, final boolean reload) – Loads the file corresponding to the buffer into the specified view. If reload is true then will not prompt the user to load an autosave version if one exists.

save(View view, String path) – saves the file corresponding to the buffer. Returns a *Boolean* which is true if the save was successful.

Attribute Methods

getMode() – Returns the *mode* for the current buffer.

setMode(Mode mode) – Sets the mode for the current buffer to ‘mode’.

getTabSize() – This method returns an *int* containing the distance between tab stops, measured in character columns. If these properties are not individually set for a specific buffer, they are inherited from the properties of the buffer's associated editing mode.

Marker Methods

addMarker(char shortcut, int pos) – Adds a marker for the line indicated by pos using shortcut. Set shortcut to ‘\0’ to indicate the absence of a shortcut.

getMarkers() – Returns a *Vector* containing the buffer's current markers.

removeMarker(int line) – Removes all markers at the specified line.

General Editing Methods

getText(int offset, int length) – This method extracts a portion of buffer text having length ‘length’ beginning at offset position ‘offset’. The method returns a newly created *String* containing the requested excerpt.

insert(int offset, String text) – This method inserts the string text at offset offset in the buffer.

remove(int offset, int length) – This method removes length characters of text starting from offset.

BUFFERHISTORY

The class BufferHistory is used to read and write to an xml file containing the recently opened files in jEdit along with other details such as the caret position and the encoding type for the file. The BufferHistory class demonstrates the Singleton pattern, since only one “history” file is required for the program. All files that require history information to be held, is held in a different entry in the “history” file.

BufferHistory contains two public classes, Entry and RecentHandler, which extends HandlerBase. An Entry contains all the required information about one file that will be stored in the “history” file (i.e. path, caret position, encoding etc).

Attributes

All attributes of BufferHistory are static.

Name	Type	Description
History	Vector	Holds all the instances of the Entry classes, i.e. the structure of the history file
Max	int	The maximum number of Entry entries in the history structure and file

Methods

load(File file) – load the “history” file putting the handle in the specified file, ‘file’.

save(File file) – saves the “history” file to the specified file, file.

getEntry(String path) – gets the Entry specified by path.

setEntry(String path, int caret, Selection[] selection, String encoding) – removes the old Entry, and adds the updated form.

addEntry(Entry entry) – adds an Entry to the history structure, and removes the first element if too many elements. This is a private method.

removeEntry(String path) – removes the Entry specified by the path. This is also a private method.

MARKER

Markers serve as textual bookmarks. A marker is a named location in a buffer, to allow easy jumping between different buffers as well as different locations within the same buffer.

Attributes

Name	Type	Description
buffer	Buffer	This is the buffer where the marker is located
shortcut	char	This stores the shortcut to the marker
pos	int	This stores where the position in the buffer is going to be when it is set.
position	Position	This is the actual position of the marker.

Methods

Constructor – The constructor is package-private and takes 3 parameters. The buffer to which the marker relates, a shortcut character, and the position to which the marker refers.

getShortcut() – this returns a *char* which is the value of the shortcut of the marker.

getPosition() – this returns an *int* which is either null or the offset of the position.

setShortcut(char shortcut) – it sets the marker’s shortcut to the value specified.

createPosition() – creates a Position at the place which is specified by the private variable *pos*, and sets the private variable ‘position’ to that new Position.

removePosition() – this sets the value *pos* to the offset of *position* and then it removes the position.

setPosition(int pos) – the allows the caller of the method to specify where the position is to be. This method should be called before createPosition() if a specific position is needed.

MODE

This is a public class. An instance of this class is created for each supported edit mode. An edit mode defines the specific settings for editing some type of file.

Attributes

Name	Type	Description
name	String	This is the name of the mode. It is used when listing the modes and when queries the properties of the mode
props	hashtable	This is where all the properties of the mode are stored
firstlineRE	RE	This is a “regular expression” of the first line property
filenameRE	RE	This is a “regular expression” of the filename
marker	TokenMarker	This is the token marker associated with the Mode. It can split lines of text into tokens. Each tokens have properties such as colour and font style.

Methods

Constructor – The constructor that takes a *String*, which is the name of the mode. It also creates a hashtable to store the properties of the mode in.

init() – initialises the edit mode. It should only be called after all the properties have be loaded and set.

getTokenMarker() – returns the *TokenMarker* marker.

setTokenMarker(TokenMarker marker) – sets marker to whatever is passed to the class.

getProperty(String key) – returns an *Object* of the property indexed in the hashtable by ‘key’.

getBooleanProperty(String key) – returns a *boolean* that is the value of the property indexed by ‘key’. It is similar to the method above.

setProperty(String key, Object value) – stores a property of the mode in the hashtable. The given value is stored under the property name given by ‘key’.

unsetProperty(String key) – this removes the specified property from the mode by deleting the entry in the hashtable under the key that is given.

setProperties(Hashtable props) – this sets the properties of the mode to an already existing hashtable of mode properties.

accept(String fileName, String firstLine) – This returns a *boolean* which is true if the edit mode is suitable for editing the given file.

getName() – returns the name of the edit mode, which is a *String*.

toString() – returns a *String* representation of the edit mode.

REGISTERS

Registers are something in jEdit that are similar to a clipboard. It is a place where text is stored so that it can later be put elsewhere. It has methods such as cut, copy and paste.

The Register class also contains several inner interfaces and classes.

The sub-interface Register is a template of a single register that is designed to store one string. It contains a method to represent the register as a String and a method to set the register contents with the passed String.

The class also contains the subclasses ClipboardRegister and StringRegister. Both implement the Register interface. The former contains an attribute, clipboard (of type clipboard) that

acts in much the same way as a clipboard in Word, while the latter is a class that holds a string, which is able to be retrieved and set.

Attributes

Name	Type	Description
registers	Register[]	This is an array of places for text to be stored

Methods

copy(JEditTextArea textArea, char register) – this is a static method which copies the text in the given text area to the specified register.

append(JEditTextArea textArea, char register, String separator, boolean cut) – there are three append methods. The other two do not have as many parameters and just use default values of the missing parameters. This method adds the text in the text area to the given register separating what is already there with a separator. If the boolean value, cut, is true, the text in the text area is removed.

cut(JEditTextArea textArea, char register) – this static method removes the text from the text area and puts it in the given register. However if the text area cannot be edited, it beeps and does not do anything.

paste(JEditTextArea textArea, char register, boolean vertical) – puts the text that is in the register into the text area. However if the register is empty or the text area cannot be edited it beeps. If vertical is true the text is added vertically. There is another paste method but does not have the boolean parameter, it just calls this method and sets vertical to false.

getRegister(char name) – returns the *Register* that is stored under the given name.

setRegister(char name, Register newRegister) – adds the new register to the array of registers under the given name.

clearRegister(char name) – sets the register of the specified name to be empty.

getRegisters() – returns the *array of Registers*. These are all of the available registers, some of which may be empty.

TEXTUTILITIES

This class contains static methods that may be useful in handling buffer text, such as finding matching brackets, and wrapping text. It includes some methods for text manipulation and finding particular positions.

Attributes

There are no important attributes in this class. There are some private static final variables that hold constants to make the programming easier, but that is all.

Methods

getTokenOffset(Token tokens, int offset) – returns the *Token* from the token list given as a parameter that contains the given offset.

findMatchingBracket(Buffer buffer, int line, int offset) – returns an *int*, which is the offset of the matching bracket or -1 if the character is not an integer or there is no matching bracket.

findWordStart(String line, int pos, String noWordSep) – returns an *int* that corresponds to the start of the word in the given String at the position pos. noWordSep is a list of characters that are not alphanumeric but are still counted as part of the word if found.

findWordEnd(String line, int pos, String noWordSep) – similar to findWordStart() but it returns an *int* corresponding to the position of the last character in the word.

getBooleanProperty(String key) – returns a *boolean* that is the value of the property of the given name. It is similar to the method above.

regionMatches(boolean ignoreCase, Segment text, int offset, char[] match) – returns a *boolean* stating whether a given segment of text matches the given array of chars at the offset. It can be used either as case sensitive or not.

spacesToTabs(String in, int tabSize) – converts consecutive white spaces to tabs of the given tab size.

tabsToSpaces(String in, int tabSize) – converts tabs to white spaces

format(String text, int maxLineLength) – this returns a formatted *String*. It makes sure that no line is longer than the maxLineLength.

getStringCase(String str)– returns an *int* representing whether a String is mixed case (0), all lowercase (1), all uppercase (2) or title case (3).

toTitleCase(String str)– returns the *String* given but changed into title case.

MISCUTILITIES

This class contains some useful static methods related to the utility of JEdit. There are methods to do with the path, text methods, sorting and comparing methods as well as some more miscellaneous methods. This class contains a lot of methods so only the most useful have been included.

Attributes

This class has no attributes. This is probably because most of the methods are not related to one another.

Methods

getFilename(String path) – this returns a *String* which is the filename in the given path.

getFilenameNoExtension(String path) – returns a *String* that is the filename without the extension included.

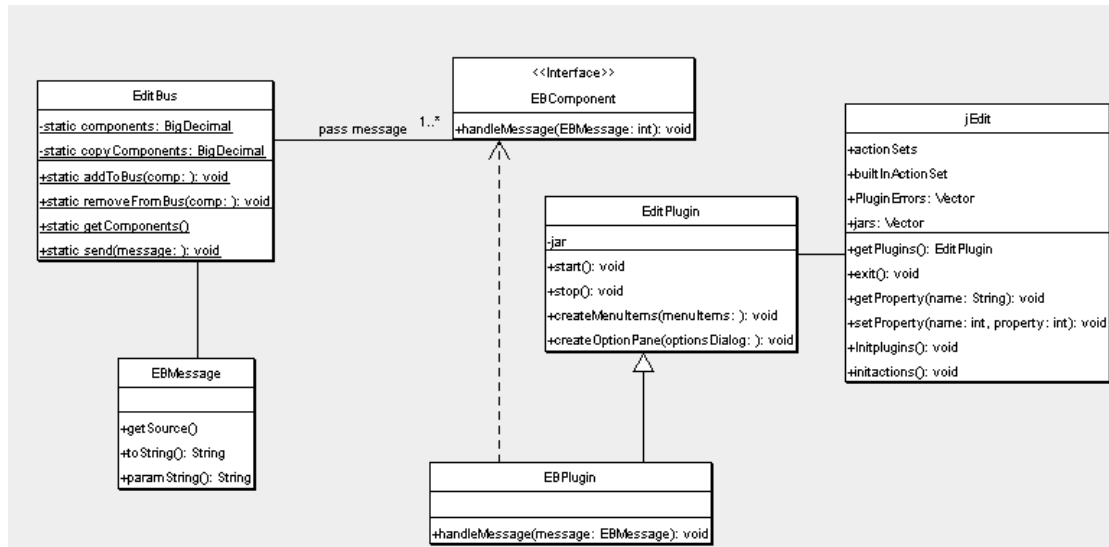
getFileExtension(String path) – returns the *String* of the extension of the file.

getParentOfPath(String path) – returns a *String* which is the directory of the specified local file.

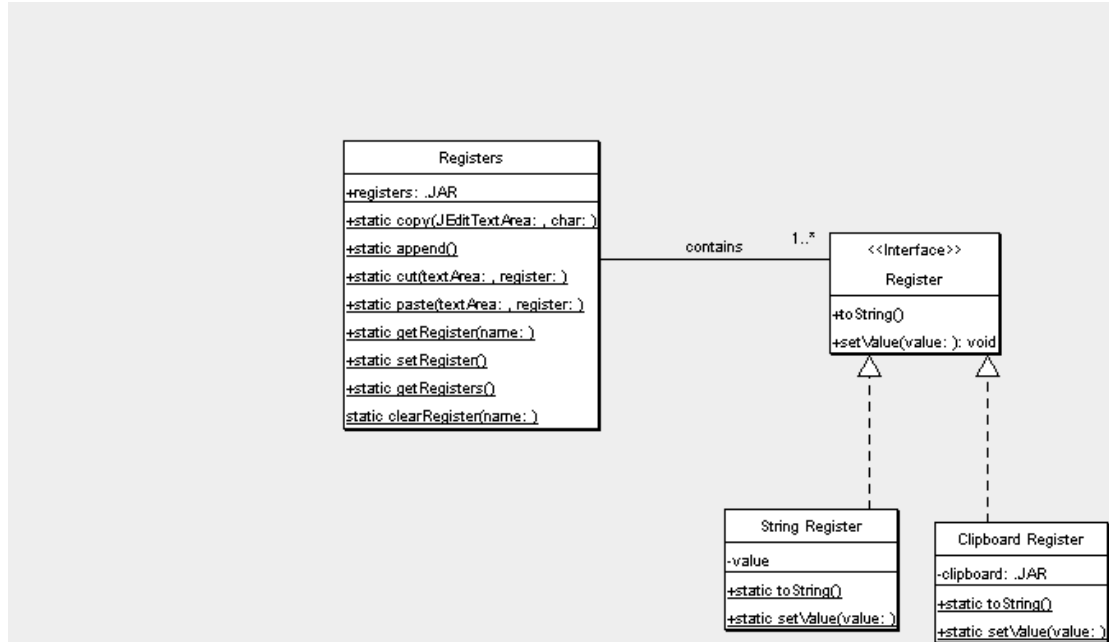
constructPath(String parent, String path) – creates and returns a *String* that is an absolute directory of the path and the parent directory. If the path is already absolute it returns that otherwise it creates one from the parent. If the parent is null the current directory is assumed.

compareStrings(String str1, String str2, boolean ignoreCase)– returns an *int* meaning either $str1 < str2$ (-1), $str1 = str2$ (0) or $str1 > str2$ (1).

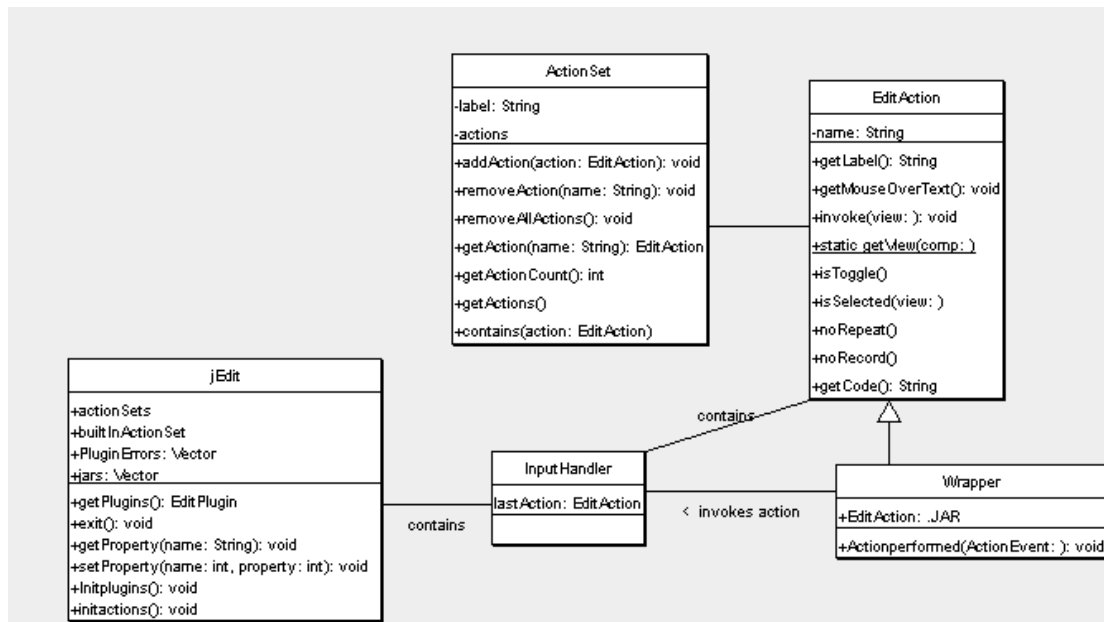
UML



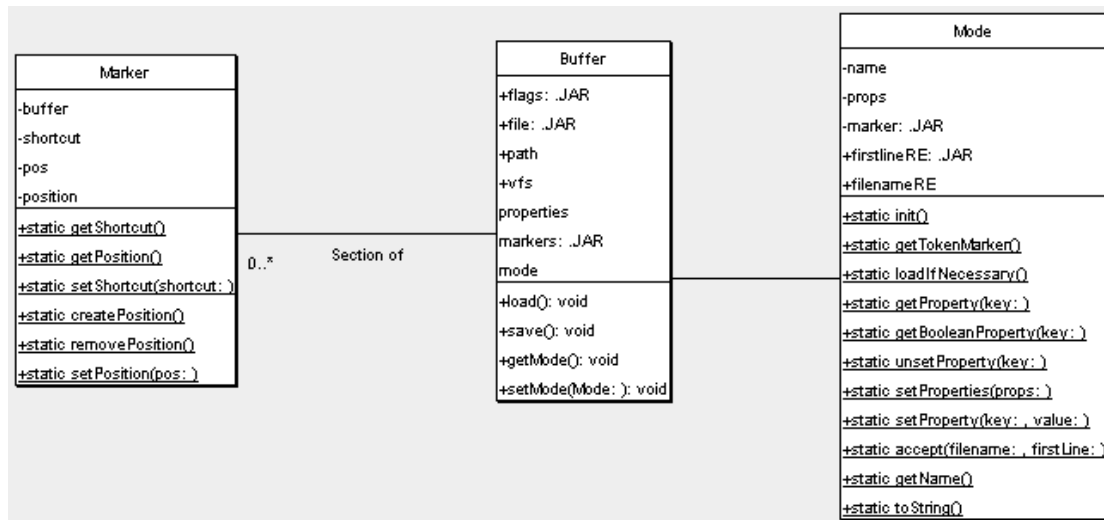
This shows the relationship between the EditBus class and the EBPlugin class, which implements the EBComponent interface. The EditBus class acts somewhat similarly to the Mediator pattern, where the EBComponent and EBPlugin classes act in a similar role to the Colleague and Concrete Colleague respectively, and the EditBus class acts like the mediator, passing messages between the various Plugins.



The Registers class contains a collection of objects which implement the register interface. String Register and Clipboard Register implement this interface.



This diagram shows the interaction between The InputHandler class acts in a similar role to that of the controller, handling actions passed to it by the wrapper class. These actions take the form of EditAction objects. (although this is an abstract class and therefore cannot be directly instantiated)



Marker represents a section of a buffer, by referring to the buffer and giving a location within that buffer. There can be many markers referencing a buffer. Each buffer must also have a Mode.

Patterns

A “software pattern describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution”. There are a few patterns that can be observed in the jEdit source code. There may be slight differences between the relationships observed and the actual pattern. The patterns found were generally types of responsibility patterns. These change the normal balance of responsibility and concentrate the responsibility in a central area.

Two of the classes analysed were example of the singleton pattern. This pattern ensures there is only one instance of the class created and is implemented using static variables and methods. No instance variables are contained within the class. The EditBus and BufferHistory exhibit this behaviour. As the EditBus is the central point of communication between the components of jEdit it is necessary that only one EditBus object exist. Similarly, the BufferHistory is a repository that stores information about recently opened files. Therefore it is also required to be a singleton class.

The EditBus and its surrounding related classes also seem to exhibit a design somewhat similar, but not exact, to the Mediator pattern. This pattern facilitates the interaction between objects through a central class. In line with this, the EditBus provides a centre through which EBComponents are able to communicate to each other by sending messages. The EditBus could be compared to the role undertaken by the Concrete Mediator however rather than coordinating the EBComponents themselves it coordinates the sending of the message objects between them, as received from other objects. The EBComponents interface would serve as the Colleague and is implemented by various classes such as EBPlugin, which act as the Concrete Colleague. However there is no Mediator interface which the EBComponents can communicate with to the Concrete Mediator through. All communication to the EditBus involves the direct calling of the class' static methods.

jEdit seems to implement a pattern similar to the controller pattern through the editAction, wrapper and inputHandler classes, where the inputHandler class in the GUI package, (which we did not look at in a lot of depth) fulfils the controller role. It handles ActionPerformed() messages sent to it by the Wrapper class (possibly among others, but that was the only one that we discovered)

The editAction class is an abstract class, which describes the Actions to be performed. The Wrapper class inherits from EditAction giving an implementation for it, which importantly, sends actionPerformed() messages to the inputhandler, alerting the InputHandler that the action has been taken.

I think that this is quite an unusual implementation of this pattern, and find it quite difficult to understand.

References

D Carrington, *Comp2501 Lecture Notes*, Brisbane, QLD, 2002.

Andy Hunt and Dave Thomas. Software Archaeology, *IEEE Software*. 19(2): 20-22, March/April 2002.

C. Larman. *Applying UML and Patterns*. 2nd Edition, Prentice Hall, 2002

Java Sun web page, java.sun.com, 12/09/2002

Java World web page. www.javaworld.com, 12/09/2002

jEdit Website, www.jEdit.org, 12/09/2002

jEdit Help facility.