Spell-As-You-Type Spell Checking in Java Swing

By: Steven Lawrance, slawrance@yahoo.com

Draft Presentation Outline

1. Purpose
2. Motivation
3. Architectural Overview
4. SpellChecker Interface
5. AspellSpellChecker
6. CachedSpellChecker
7. Remote SpellChecker Over CORBA
8. SpellCheckerContent
9. SpellCheckerThread
10. SpellCheckerManager
11. Swing Extensions
12. Tying it All Together - Demo
13. SourceForge Project Page (not set up yet, but will be well before JavaOne)
14. Future Directions

Purpose

Spell-as-you-type spell checking in Swing enhances an application's usability while ensuring the correct spelling of user-entered text in implementing applications. By underlining misspelled words as they are typed in, users get immediate feedback on their misspellings just as they do in popular word processors.

Depending on the nature of the application, as one hypothetical scenario, user-entered text in a database could get submitted as evidence in a court, and misspelled text from a database could embarrass an organization, even if to a minor extent. Reviews and audits for compliance or routine monitoring share similar implications.

Although this may be far-fetched, I'm hoping that this framework or a derivative thereof finds itself in the standard Java class libraries in addition to the native Aspell library, though a pure-Java spell checker will also work in lieu of Aspell. This can bring standardized spell checking to all Java platforms, not just Mac OS X.

Motivation

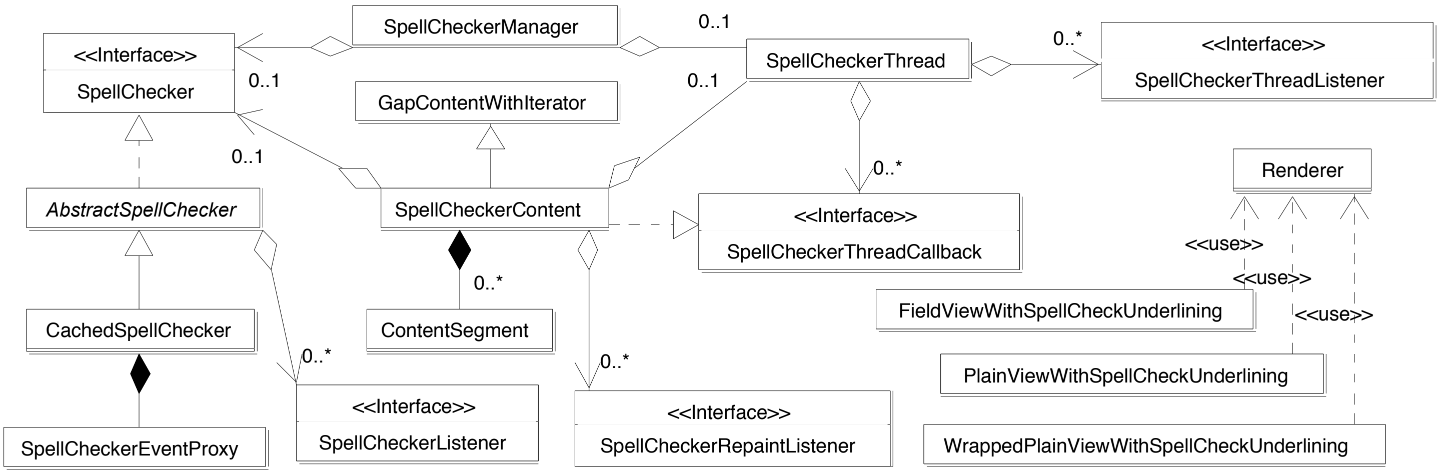
Users of a multi-organization shared client registration system that I develop for originally typed their clinical and psychosocial notes into a Microsoft Word window for its spell checking and then copied the results into the application's note entry screens. While this may work well in many organizations, we found that checking the spelling of medical terms with accurate word suggestion lists using Microsoft Word would likely add high acquisition, deployment, and maintenance costs to the project as we would need to purchase and update specialized medical dictionaries for all workstations. Centralizing the word lists could solve this problem, though that meant that, at the time that we looked at it, using Word was out of the question.

Seeking to streamline the process of entering notes while strengthening the spelling of our notes, the organizations involved prioritized the addition of a spell checker, giving the green light to this project. Seeking maximum return on minimal investment, I chose to reuse the open-source GNU Aspell Spell Checker, regarded as the best spell checker out there in terms of its word replacement suggestions. To solve the word list centralization and deployment issues, I used CORBA to send spell checker method calls to the spell checking server, where JNI can call native Aspell functions.

After about two years of reliable spell checking within the deployed application and with my graduate school term set to start in the Fall of this year, I decided to write this paper describing this framework and release it under an open-source license so that I and others can use and improve it going forward.

Architectural Overview

Seeking a solid abstraction that does not tie the framework to one or one set of spell checkers, I designed the spell checker framework with multiple interfaces to permit interchangeable implementations.



At the forefront, the SpellChecker interface presents spell checking algorithms and libraries in a standardized manner for applications and the rest of this framework. The AbstractSpellChecker implements the SpellChecker interface to ease a full SpellChecker's implementation and is used by the CachedSpellChecker, which delegates spell check requests to another SpellChecker while caching words that are deemed as correctly spelled to enhance the performance of future word checks on slow or network-based SpellCheckers.

SpellCheckerContent instances get used by AbstractDocuments to add misspelled word tracking on the content layer, ensuring that the SpellCheckerContent is aware of all content mutations. This paper discusses the rationale for operating on the Content layer rather than the Document layer and poses a migration to the Document layer as a potential future enhancement. Applications set the “content” property on a JTextComponent when the CustomTextAreaUI and CustomTextFieldUI look-and-feel extensions, explained later, are used.

The SpellCheckerThread class manages the background checking of SpellCheckerThreadCallback-implementing objects such as the SpellCheckerContent. SpellCheckerThreadListeners can receive notifications on when spell checking begins, when it finishes, when spell checking will likely take more than three seconds to finish, and when the SpellChecker in use throws an exception, giving useful feedback to desktop applications and applets.

The CustomTextAreaUI and CustomTextFieldUI classes use the FieldViewWithSpellCheckUnderlining, PlainViewWithSpellCheckUnderlining, and WrappedPlainViewWithSpellCheckUnderlining classes to draw spell-checked text with misspelled word underlinings by delegating their drawSelectedText() and drawUnselectedText() methods to the framework's Renderer class, which queries the proper SpellCheckerContent for misspelled word ranges.

Swing's excellent extensibility made this framework possible and, through this extensibility, this spell checking framework can likely be plugged into most Swing applications without too much work, especially applications that already use FieldView, WrappedPlainView, or PlainView for their JTextComponent views. Applications that use custom Views in their TextUIs might require additional integration work for the spell-as-you-type underlining.

In summary, this framework's core objects are the SpellChecker interface, the SpellCheckerContent, the SpellCheckerThread, and the Renderer. Each has its own purpose, and they all tie together through abstraction to produce spell-as-you-type spell checking with Java Swing.

SpellChecker Interface

<<Interface>> SpellChecker
getCapabilities() : int
isCapabilitySupported(capability: int) : boolean
saveAllLists() : void
addSpellCheckerListener() : void
removeSpellCheckerListener() : void
getSpellCheckerListeners() : SpellCheckerListener[]
checkSpelling(word: String) : boolean
getWordSuggestions(misspelledWord: String) : String[]
setWordReplacement(misspelledWord: String,correctSpelling: String) : void
addInstanceWord(word: String) : void
addPersonalWord(word: String) : void
addDictionaryWord(dictionary: String,word: String) : void
removeInstanceWord(word: String) : void
removePersonalWord(word: String) : void
removeDictionaryWord(dictionary: String,word: String) : void
getInstanceWords() : Set
getPersonalWords() : Set
getDictionaryWords(dictionaryName: String) : Set
clearInstanceWords() : void
clearPersonalWords() : void
getPropertyDescriptor(key: String)
setPropertyValue(key: String,value: Object) : void
getPropertyValue(key: String) : Object
getProperties() : Map
setProperties(map: Map) : void
getSpellCheckerName() : String
getSpellCheckerVersion() : String
finalize() : void

The SpellChecker interface, depicted on the left, includes various methods related to spell checking along with event notification and property methods. An abstract implementation, the AbstractSpellChecker, provides empty implementations for these methods except for several that all spell checkers must implement: checkSpelling(), getCapabilities(), getSpellCheckerName(), and getSpellCheckerVersion(). Other methods are optional, though a fully-featured spell checker will implement most or perhaps all of them.

Beyond the simple ability to check the spelling of a word and return a binary-ORed integer of capability flags, spell checkers may optionally implement word replacement suggestions, SpellChecker object instance-specific words, a personal dictionary, generic named dictionaries, setting custom misspelled word replacements, list saving, word list retrieval using the Collections Set interface, SpellChecker-specific configuration properties, and public finalization.

In terms of how popular word processors operate, word replacement suggestions are the word lists that appear when a user right-clicks on a misspelled word or comes across a misspelled word in a spell checking dialog box. The getWordSuggestions() method takes a misspelled word as its argument and returns a String array sorted from the best suggestion to the worst suggestion or, if no suggestions exist, an empty String array.

SpellChecker object instance-specific words refer to words that persist only as long as a SpellChecker object instance is alive or until clearInstanceWords() gets called. This is analogous to the “Ignore All” command in a spell checker user interface that tells the spell checker to ignore all instances of a given misspelled word. Words get added through addInstanceWord() and removed through either removeInstanceWord() or clearInstanceWords(). Through the magic of event notifications, an “Ignore All” command on a word in one text box can automatically remove the misspelled underlinings for that word in all other text boxes that use the same SpellChecker instance, and that is exactly what the SpellCheckerContent object does, among other things.

Personal dictionaries are a bit self-explanatory as most word processor spell checkers implement them. Rather than clicking “Ignore All,” a user can add a misspelled word to a personal dictionary to permit that word to persist across SpellChecker instances. Most word processors call this function “Add Word.” In the SpellChecker interface, addPersonalWord() adds words to this list. The particular personal word list file or resource that a SpellChecker instance uses is an implementation-specific property as not all SpellCheckers are necessarily local or even use a filesystem as a back-end store.

In addition to personal lists, the SpellChecker interface lets applications manipulate named dictionaries such as master or supplementary word lists. The generic addDictionaryWord(), removeDictionaryWord(), getDictionaryWords, and clearDictionaryWords() methods grant access to these dictionaries. As these methods are rather new and aren't used by the client application that I work on, more general dictionary methods will likely get added such as dictionary enumeration, creation, removal, loading, and unloading methods. Perhaps a new dictionary descriptor class is in order. This and other possible enhancements are discussed later in this paper.

Users who commonly misspell certain words may tell their word processor to “Always Replace” a particular misspelled word with the correctly-spelled word. The setWordReplacement() method accomplishes this and lets the SpellChecker implementation factor that into future calls to getWordSuggestions(). Setting a word replacement should persists across SpellChecker instances that use the same word replacement lists in a manner similar to how personal words are handled, though that is not absolutely required.

The saveAllLists() method instructs the SpellChecker to write any unsaved or in-memory changes to objects such as personal word lists or word replacement lists to the persistent storage. In many cases, this simply translates into writing the changes to files. This method does not guarantee that it was physically written to the underlying media, but a successful return does guarantee that the lists were saved such that new SpellChecker instances that use the same files or resources will contain the same word list data. This method will probably have an exception type added to its list of exceptions to propagate failure events to the caller.

Applications that wish to retrieve dictionary, personal, or instance word lists may call getDictionaryWords(), getPersonalWords(), or getInstanceWords(), respectively, in the form of a Java Collections Set. The order of the Set is defined by the SpellChecker implementation and might not appear in alphabetical order to an Iterator.

SpellChecker implementations may optionally support property enumeration, assignment, and retrieval through the SpellChecker's property methods. Applications can retrieve property metadata through either the getPropertyDescriptor() or getProperties() method and thus receive a property's description, read-only versus read-write status, default value, name, and current value along with a method that can set that property's value. This amount of information should let an application present a list of SpellChecker configuration properties to a user. The setPropertyValue() and setProperties() methods can be used to change property values and generate property change events to registered listeners after each property change takes place. The setProperties() method accepts any object for Map values and, if a value is an instance of SpellCheckerPropertyDescriptor, the value within that SpellCheckerPropertyDescriptor is used, allowing an application to call this method using a Map returned as-is from the same or another SpellChecker instance's getProperties() method call. Not all spell checkers support on-the-fly property changes for some or all properties, so changing a property may cause a spell checker reset event to fire, indicating the clearing of all instance words and possibly the loss of any unsaved word or replacement list modifications.

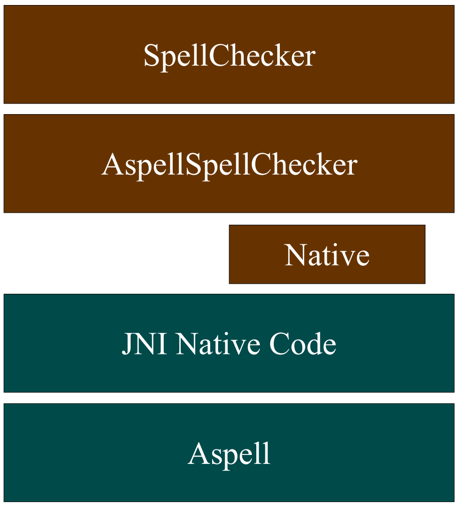
One may notice that the SpellChecker interface includes a public finalize() method that publicizes Object's protected finalize() method. While I'm not yet sure if this is the best approach to this or not, some SpellChecker implementations such as the AspellSpellChecker use native objects over JNI, and having the ability to explicitly deallocate native Aspell objects in memory can help keep native Aspell memory use in check. I've noticed that finalize() is not always called reliably on some JRE versions, so having a public finalize() method helps ensure that an application can explicitly free native objects on SpellChecker objects that utilize JNI. Of course, simply calling finalize() doesn't garbage-collect the object at all, and finalize() in SpellChecker implementations must not assume that they will only get called once. I welcome input on how to handle this and whether or not a JNI class can rely upon finalize() getting called during garbage collection.

Because not all SpellChecker implementations are the same, each one must return a list of supported SpellChecker capabilities through the getCapabilities() method, which binary-ORs integer constants representing that SpellChecker's capabilities. Applications should call getCapabilities() or isCapabilitySupported() when setting up their spell checker user interfaces so that only supported capabilities are exposed. Capabilities can also dynamically change in a SpellChecker. When that happens, registered listeners receive a “capabilities changed” event. As an example of how an application adjusts a user interface to match a SpellChecker's capabilities, SpellCheckers that support personal word lists can enable an “Add Word” button or right-click menu item while SpellCheckers that don't support personal word lists can gray or hide that button or right-click menu item.

Applications can easily use implementations of this interface as-is without the spell-as-you-type part of this framework. Some examples could include search engines that check the spellings of queries and recommend correctly-spelled queries, dialog-based spell checking on documents, electronic dictionary or encyclopedia applications, or even simple dedicated spell checking applets. Of course, the main focus of this framework is the spell-as-you-type checking, so read on to see how this leads up to Swing integration.

AspellSpellChecker

The Aspell SpellChecker implementation provides a Java wrapper to the native C-based Aspell spell checker over JNI. As Aspell was the originally-implemented SpellChecker object, much of the SpellChecker interface resembles the general design of the Aspell API, though not all methods in SpellChecker are supported by Aspell. The AspellSpellChecker object presents itself as a SpellChecker implementation extending the AbstractSpellChecker while relying upon inner classes for native methods, Sets, Set Iterators, and property descriptors. To the maximum extent possible, complexity is kept in the Java implementation while the native JNI code simply acts as a dumb bridge between the Native inner class and Aspell's C API.



Because Aspell's C API includes functions for enumerating lists and querying metadata dynamically, very little impedance mismatch exists between Aspell's C API and the SpellChecker interface. As an example, the word list retrieval methods such as getPersonalWords() and getInstanceWords() return a Set implementation that defers its methods calls to the appropriate C-based methods in the Aspell API, including the Iterator class and methods. No copying to a Java-backed Set happens in those two methods.

Hopefully, one day, Aspell will have a pure-Java implementation and thus remove the need for a JNI bridge to use it. Aspell's implementation utilizes object orientation to a high degree, though, like many porting projects, it comes down to time and interest, and using the native Aspell library through JNI works well for now. I successfully built and used the native JNI code to AspellSpellChecker on Windows, Linux, and Mac OS X, and chances are very good that it will also work on Solaris. Pointers originally used 32-bit integers, though now the JNI bridge uses 64-bit pointers between C and Java while casting appropriately for Aspell pointers depending on whether the native platform's processor and operating system are 32-bit or 64-bit.

Because SpellChecker is an interface rather than an implementation, one certainly does not need to use the AspellSpellChecker to take advantage of this framework. I mentioned the AspellSpellChecker because it is currently the fullest working implementation of the SpellChecker interface at this time.

CachedSpellChecker

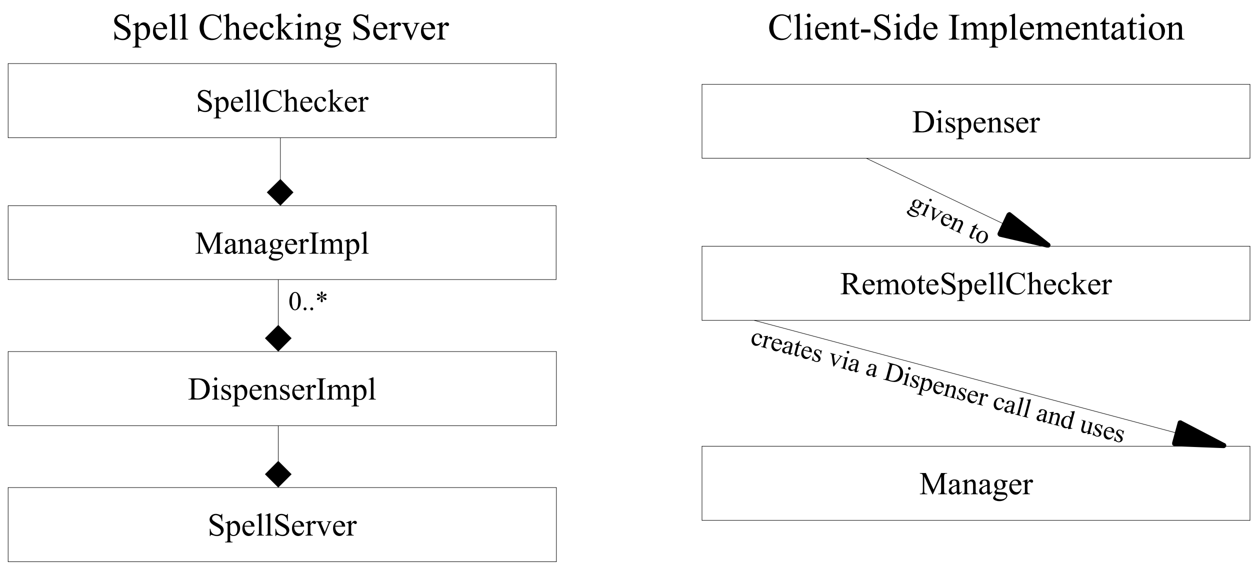
The CachedSpellChecker maintains a list of correctly-spelled words to accelerate calls to checkSpelling() while delegating all other methods and checkSpelling() calls on unknown words to a delegate SpellChecker object. Events generated by the delegate SpellChecker find their way to a CachedSpellChecker's listeners through an event proxy. When used with network-enabled or otherwise slow-running SpellCheckers, the CachedSpellChecker offers spell checking services at a reduced network and/or computation cost.

The CachedSpellChecker actually maintains two lists: a list of correctly-spelled words and a list of instance words. As instance words have a lifetime of only a SpellChecker object's instance, the CachedSpellChecker can provide instance word services for SpellCheckers that don't support instance words in addition to SpellCheckers that do support instance words. It actually provides this service regardless of underlying support, though if the delegate SpellChecker does support instance words, the CachedSpellChecker will initialize its instance word list at the time that SpellChecker is set as the delegate, adding those words to the overall cache in the process. All instance words exist in the overall list of correctly-spelled words, and this is maintained through the relevant method implementations and event handlers (e.g., adding an instance word to the delegate SpellChecker directly rather than through the CachedSpellChecker will get picked up by the CachedSpellChecker).

In addition to implementing instance words, the CachedSpellChecker can run in a standalone mode without a delegate SpellChecker, though with limited functionality. Methods exist for reading its main word list cache from a Reader and writing it out to a Writer using a simple text file format of one word per line (blank lines are ignored). The CachedSpellChecker also implements Serializable to support saving and loading the main word list cache through Java's serialization capabilities.

Remote SpellChecker Over CORBA

The client application that I write for presently uses CORBA for its client-to-server communication and influenced the remoting technology chosen for a network-enabled SpellChecker. CORBA is not central to the implementation's design; one can substitute RMI or Web Services with a little refactoring. The overall design of the remote SpellChecker utilizes a simple and effective dispenser and manager implementation on the server side with a small client-side object that implements SpellChecker.



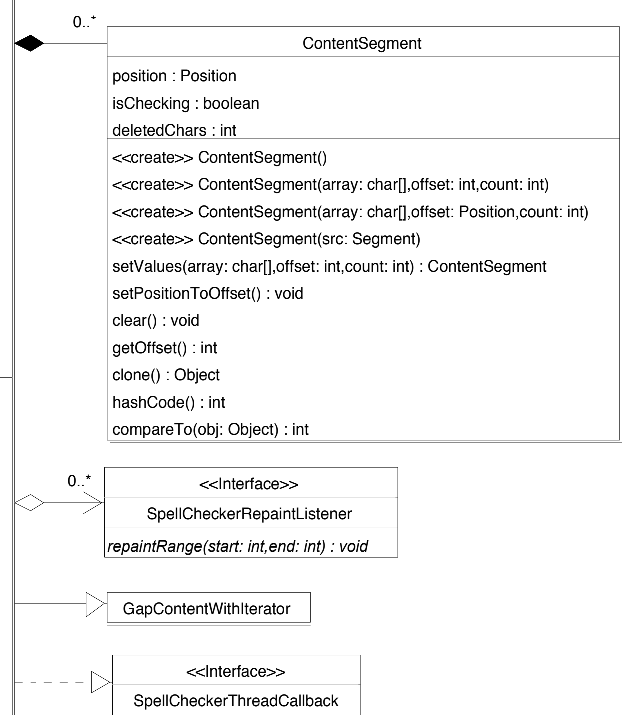
On the server side, the SpellServer creates a DispenserImpl instance and publishes it on a transient name server. The DispenserImpl exposes only two methods: one for creating a new ManagerImpl and another for retrieving the inactivity timeout. The DispenserImpl closes ManagerImpl objects and, consequently, SpellCheckers when the client explicitly requests a closure or when the client has not performed any activity during the timeout period. The client regularly calls a keepAlive() method in the DispenserImpl for the lifetime of the RemoteSpellChecker. Each client RemoteSpellChecker instance interacts with its own corresponding ManagerImpl on the Spell Checking Server, and each ManagerImpl calls methods on its own SpellChecker instance. At this time, events from the server-side SpellChecker do not propagate to the RemoteSpellChecker, so the RemoteSpellChecker fires its own events after calling remote Manager methods.

The client application that I write for uses this spell checking server with the AspellSpellChecker on the server side and a CachedSpellChecker wrapping the RemoteSpellChecker on the client side to enable efficient and accurate distributed spell checking with minimal network utilization. Medical words are centrally stored on the server and managed from the front-end application in an administrative interface. Along with this setup, a personal word replacement list file exists for each user on the server through Aspell to personalize the misspelled-word-to-correctly-spelled-word mappings for each user's peculiar word misspellings. Personal word lists are not utilized on a per-user basis but rather in a application-wide manner to store the medical words, and only administrative users in the application can modify this list, though technically nothing other than Java and CORBA development abilities stops users from modifying the application-wide personal word list through direct usage of the SpellServer's Manager interface from their workstations as the security is in the application's client layer.

Interesting potential utilizations of the RemoteSpellChecker and SpellServer exist and include, but are certainly not limited to, adding centralized and stronger spell checking to Microsoft Word, creating a spelling provider for Mac OS X's spell checking framework, integrating this framework into OpenOffice.org and StarOffice, and building a text editor applet or Java WebStart application that utilizes centralized spell checking on a server. The abstraction involved permits implementors to choose which SpellChecker to use, so use of this framework does not tie people to Aspell.

SpellCheckerContent

The SpellCheckerContent checks an AbstractDocument's spelling in real-time by directly handling the insertString() and remove() methods in AbstractDocument.Content through its extension of GapContentWithIterator, which extends GapContent, and posting “unchecked range” events to a SpellCheckerThread, which will call SpellCheckerContent's checkSpelling() method from another thread using a LIFO queue of SpellCheckerThreadCallback-implementing objects that have unchecked ranges. Applications can use SpellCheckerContent's methods to query information on misspelled words in addition to forcing current word checks, canceling background checks, queuing the entire content for spell checking, setting misspelled words as correctly spelled throughout the content, performing a blocking check, and ignoring specific misspelled word instances. The CustomTextUI class, discussed later, registers repaint listeners with SpellCheckerContent objects so that the UI can efficiently repaint words as their correctly-spelled or misspelled status changes from one to the other without having to repaint the entire UI object.



To maximize code reuse and minimize complexity, the SpellCheckerContent object heavily utilizes the BreakIterator in the java.text package to determine word boundaries in the content while presenting the content as a CharacterIterator for minimum memory utilization and maximum performance. Needless to say, this approach took some time and experimentation to arrive at, but in the end, the integration of the BreakIterator with GapContent through the GapContentWithIterator extension works very well and should theoretically work with determining word boundaries on complex languages such as Japanese. Perhaps a future version of GapContent can support the creation of CharacterIterators directly, though that's another topic in itself.

This class makes heavy use of the Position objects returned from Content's createPosition() method for mutation-agnostic content positions within the ContentSegment class, which extends Segment and serves as a vehicle to inform the CustomTextUI or any interested applications of all misspelled word ranges between two arbitrary content position offsets via the getMisspelledWordRanges() method. The goal is to not penalize long contents when misspelled words are added or removed and to perform with O(1) constant execution time in as many cases as it can. This stemmed from the original requirement that spell checking must not slow down the user's entry of text, and the SpellCheckerContent object achieves that goal nicely with minimal content locking and a design that does not require the update of misspelled word positions whenever the content changes. The content remains free of any locks while the SpellCheckerContent waits for its SpellChecker to check the spelling of a word, and if the content directly related to a misspelled word range currently being checked changes, the SpellCheckerContent will automatically compensate appropriately and reschedule that range's checking with the SpellCheckerThread. To minimize the spell checking load during active content mutation, a word that is in the middle of being edited will not get checked immediately unless if, with new mutations, that word is no longer the current word or if forceCheckOnCurrentWord() gets called, which the CustomTextFieldUI and CustomTextAreaUI classes call when the text component loses the current focus.

During text insertions and removals, the SpellCheckerContent intelligently manages the misspelled word ranges affected by the current mutation. Text insertions that amend, immediately precede, or occur within a misspelled word range will increase the size of that misspelled word range and queue that new text for spell checking. Text removals delete any misspelled word ranges within the text removal range, shrink the affected word range, if it exists, at the beginning and end of the removal range, and, if two misspelled word ranges existed at the beginning and end of the removed range, those two misspelled word ranges are merged together and queued for checking. Much of the complexity in the SpellCheckerContent exists in these two methods as both implement thread safety and minimal locking.

At this time, the SpellCheckerContent tightly-couples itself to the SpellCheckerThread, though the SpellCheckerThread loosely couples itself to the SpellCheckerContent object through the SpellCheckerThreadCallback interface. A future version of this spell checking framework may loosen the coupling between the SpellCheckerContent and the SpellCheckerThread to permit alternative implementations of the SpellCheckerThread. The CustomTextUI class also tightly-couples itself to the SpellCheckerContent object, so some extra abstraction is possible in this area.

SpellCheckerThread

Running as a separate thread, the SpellCheckerThread drives the background checking of SpellCheckerThreadCallback-implementing objects, which primarily means the SpellCheckerContent object described in the previous section. SpellCheckerThreadCallback-implementing objects notify the SpellCheckerThread that they have unchecked word ranges in response to content mutation events. The last SpellCheckerThreadCallback that notified the SpellCheckerThread of its unchecked ranges gets checked first under the assumption that the most-recent notification likely occurred in response to a user-initiated event such as typing into a text field.

The SpellCheckerThread also manages the countdown on the current word that the user is in the middle of editing for it to get checked after the timeout, which is three seconds since the last character typed by default. A SpellCheckerThreadCallback-implementing object in the middle of its checkSpelling() call can optionally notify the SpellCheckerThread of its progress as it checks its content and, if the SpellCheckerThread determines that the expected check time will likely take more than three seconds based on the current rate of progress, it will notify SpellCheckerThreadListeners of this event so that, if an application wishes to, it can display a UI hint to the user to alert them of this condition.

Sometimes, a SpellChecker or a SpellCheckerThreadCallback may throw an Exception during the checkSpelling() call to a SpellCheckerThreadCallback. When this happens, the SpellCheckerThread notifies SpellCheckerThreadListeners of this along with the Exception object thrown so that applications can handle this event appropriately. Along with this event, SpellCheckerThreadListeners will also receive notifications when spell checking begins, when it ends, and when it begins and ends for each SpellCheckerThreadCallback object, giving applications detailed information on the SpellCheckerThread's status. These events run within the SpellCheckerThread's thread.

SpellCheckerManager

Nothing more than a convenience class, the SpellCheckerManager provides static methods for getting and setting a ClassLoader-wide SpellChecker and SpellCheckerThread instance so that an application does not need to maintain its own references when creating new SpellCheckerContent objects for its text fields. Of course, different SpellCheckerContent objects may use different SpellCheckers and even SpellCheckerThreads, though many applications will likely share a single instance of SpellCheckerThread throughout its SpellCheckerContents and, depending on the type of application, a single SpellChecker as well. Document-oriented applications such as word processors will probably want to use a new SpellChecker instance for each document.

Swing Extensions

Although getting Swing to underline misspelled words is a bit more complex than it sounds, it is definitely possible to make it work efficiently and without elaborate hacks. Overall, this task involves drawing misspelled text with an underline, handling right-clicks on the text field or area for a pop-up menu, maintaining that pop-up menu, forcing a check on the current word when the focus is lost, handling the “next misspelled word” hotkey (Control+Semicolon), and responding dynamically to changes in the JTextComponent's content property.

Several approaches exist to achieve the actual visual underlining effect for misspelled words. One could respond to repaint events by figuring out where the UI drew the text after the fact and draw an underline underneath it, though such an approach requires exact formatting information from the UI and adds extra processing to drawing operations. Ideally, the underlining should happen at around the same time the text is drawn when the exact pixel positions are known inside the relevant drawing functions. Fortunately, Swing provides such an extension point through the View interface that the ViewFactory create() method returns, which all ViewFactory-implementing BasicTextUI descendants implement. The CustomTextFieldUI and CustomTextAreaUI objects override their super implementations' create() methods and return the appropriate View from the inclusive set of PlainViewWithSpellCheckUnderlining, FieldViewWithSpellCheckUnderlining, and WrappedPlainViewWithSpellCheckUnderlining. Each of those View implementations diverge from their super implementations' drawSelectedText() and drawUnselectedText() methods by delegating the text drawing to this spell checker framework's Renderer class's drawText() method instead of to Utilities.drawTabbedText(). The Render still uses Utilities.drawTabbedText() for correctly-spelled words, though a modified variant of Utilities.drawTabbedText() gets used for the misspelled text drawing in Renderer.drawMisspelledTabbedText(). That method underlines misspelled words with the Renderer.underlineMisspelledText() method before writing out the characters of a misspelled word to ensure that the underlining does not potentially obstruct character descenders. The default underlining color is a soft red (R,G,B is 255, 128, 128), but this can be changed by setting the misspelledWordUnderlineColor property in a JTextComponent.

As far as I am aware, no standardized right-click menu handling exists in Swing for text components, requiring an application to handle this task itself if it wishes to have this behavior. Because spell-as-you-type spell checking really needs a right-click menu for the full user experience, the CustomTextUI implements one that lists the top seven word suggestions, misspelled word operations such as “Add for Now,” “Add to Personal List,” “Ignore this Word,” and “Next Misspelled Word,” undo and redo items when an undoManager property exists, and the standard Cut, Copy, Paste, Delete, and Select All edit items. Menu handling is a big area for potential improvement in this framework and possibly Swing in general, though it's entirely possible for an application that uses its own right-click pop-up menu to query the SpellCheckerContent itself and integrate spell checking into its menu in they way that it pleases to.

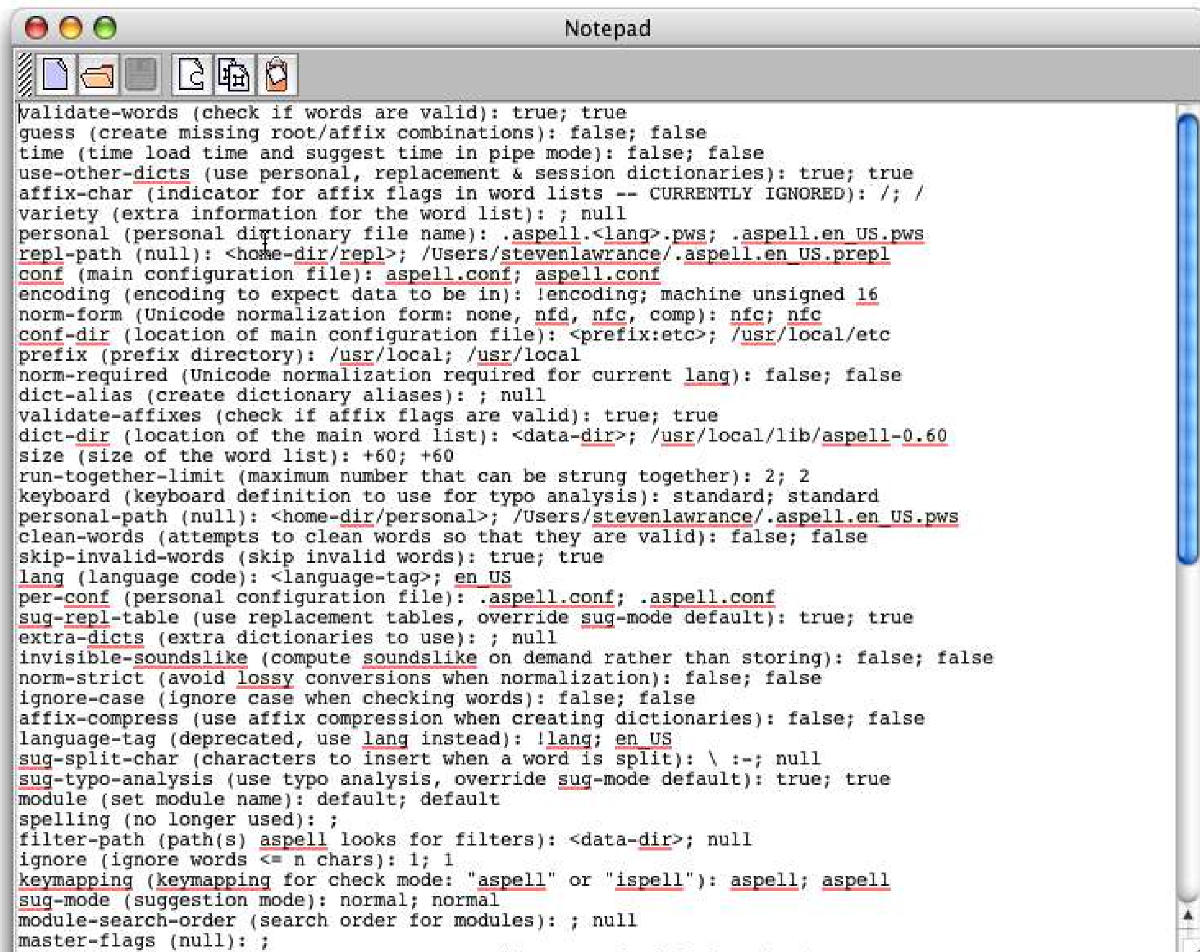
Because Apple's Mac OS X operating system defines Command+Semicolon as the hotkey that jumps to and highlights the next misspelled word, the CustomLookAndFeel uses Control+Semicolon on Windows and Linux. This can be defined or changed easily through the look and feel, especially if the CustomLookAndFeel is not used. The CustomTextUI binds the next-misspelled-word handler to the “selectNextMisspelledWord” keyboard action. The “undo” and “redo” keyboard actions bind to the undo and redo handlers that defer to the registered UndoManager in the “undoManager” property, which should be bound to Control+Z and Control+Y.

Various places in this paper refer to a content property in a JTextComponent, but no rationale for this property has been put forth yet. Why not use getDocument(), you might ask? The AbstractDocument uses a Content object for the actual text storage, which cannot be retrieved publicly from an AbstractDocument, and Content is what the SpellCheckerContent extends, but why was that approach chosen? Documents are good enough for the UI, so why doesn't the spell checking framework use them? The SpellCheckerContent needs to know about text removal events before and after the removal takes place so that it can adjust any Position objects within or adjacent to the removed text range before they become unpredictably invalidated by the text removal process. I experimented with this more than two years ago with 1.3.1_02, so perhaps this aspect is worth revisiting to see if this is still necessary in 1.4.2 and above so that DocumentEvents can be used instead of direct Content extension and thus work with a wider range of Documents. The ability to directly query the content with a CharacterIterator interface through GapContentWithIterator seemed cool at the time, though Document's Segment-based getText() method is actually just as efficient and is what the performance-demanding UI text drawing objects use anyway. A future enhancement may involve pushing the spell checking back up to the Document layer with a DocumentListener or, potentially, a delegating Document if it still must know about text removals before they happen. But for now, an application will need to set the “content” property on a JTextComponent to an instance of SpellCheckerContent to make the CustomTextUI underline misspelled words.

Tying it All Together - Demo

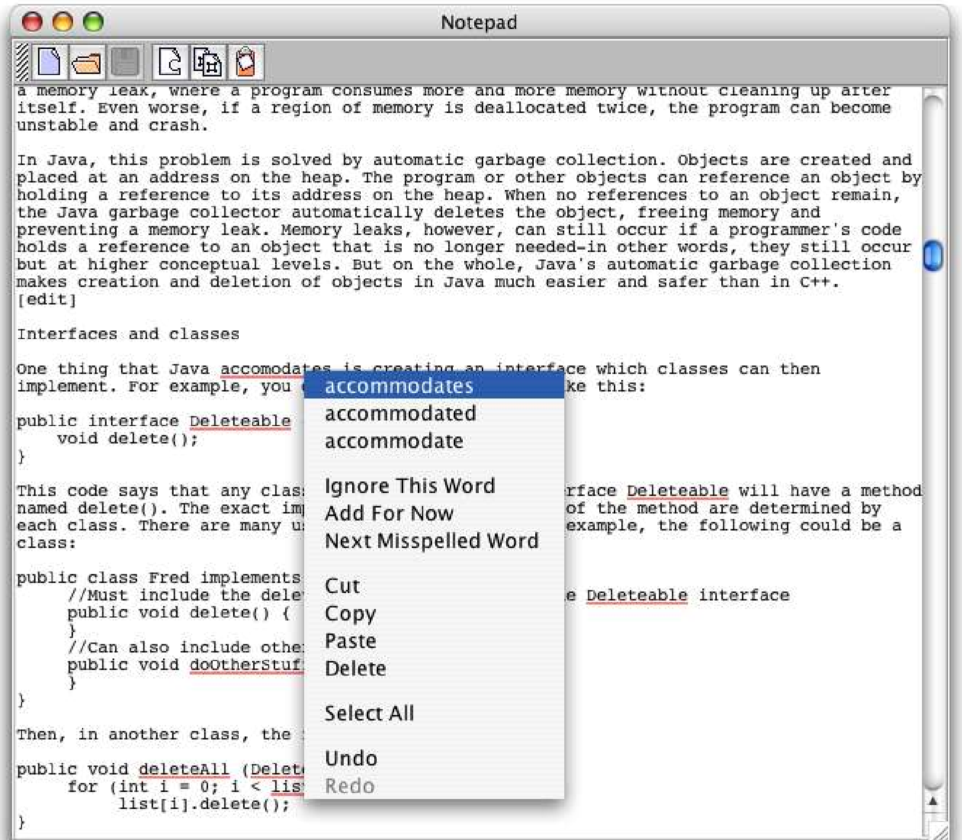
Using the Notepad demonstration program that ships with the Sun Java SDK, I added a small amount of extra code to enable spell-as-you-type spell checking to create a demonstration for this spell checking framework.

When this customized version of Notepad starts up, the initial document contains a dump of the SpellChecker's configuration from a loop that iterates through the results returned from getProperties(), as show in the image below. Each line was written in the form of [property name] ([description]): [defaultValue]; [value].

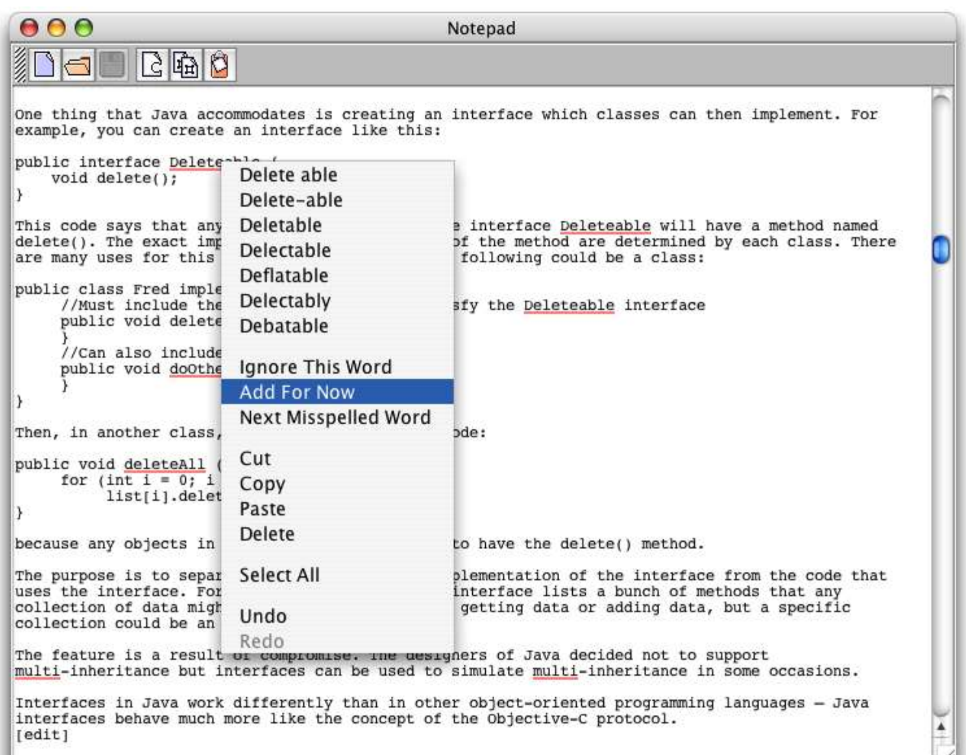


The default values work well in most cases, though applications are definitely free to adjust them as necessary, just as the application that I write for does to make the personal word list application-wide instead of per-user. Note that different SpellChecker implementations will likely support different properties. The above screen shot lists the properties in an AspellSpellChecker instance using GNU Aspell 0.60.2.

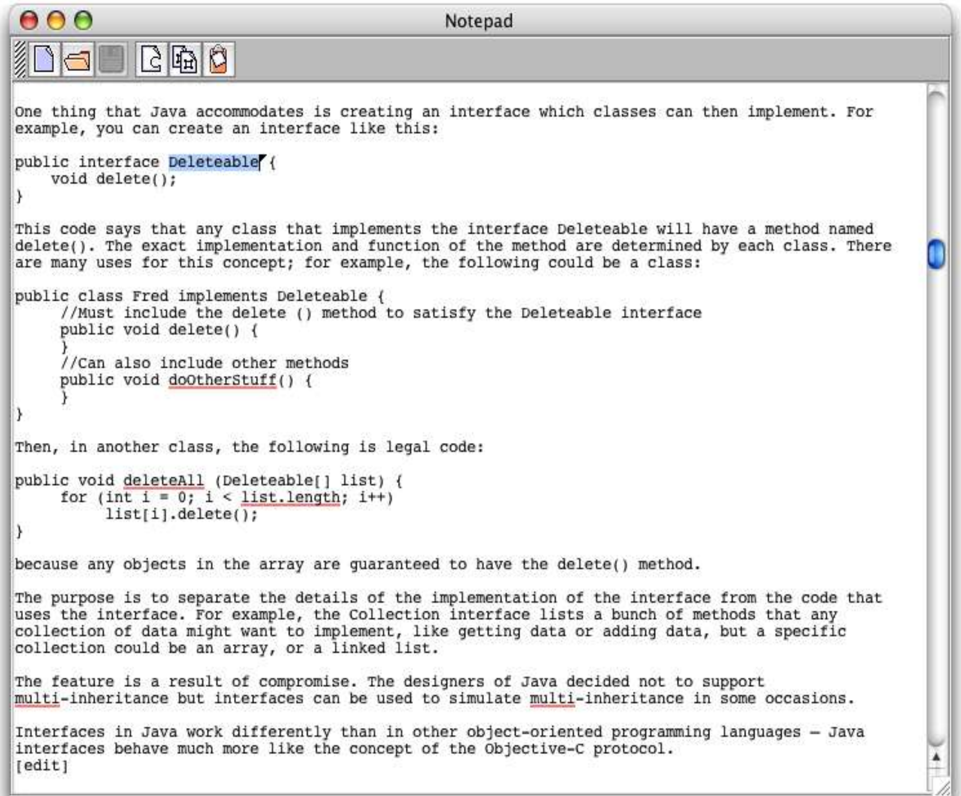
Using the Java_programming_language Wikipedia article as a test, the spell checker was able to spot misspelled words immediately after pasting the entire article into Notepad. Remember that "accommodate" is one of the few words that can accommodate two Cs and two Ms.



Using this same screen shot, notice how "deletable" is misspelled several times in the same manner. The next two screen shots demonstrate the "Add for Now" function that adds an instance word. When the SpellCheckerContent receives the instance-word-added event from the SpellChecker, it reacts to it by looking through its list of misspelled word ContentSegments and removing matching words that are now considered correctly spelled, notifying all registered SpellCheckerRepaintListeners of the now correctly-spelled ranges in the process.



Many other demonstration scenarios exist, though it's far easier to demonstrate in person than on paper, especially aspects that rely on timing. At this time, the spell-as-you-type functionality behaves much in the same way as it does in popular word processors, though no standardized spell checking dialog box exists at this time, which might get added as a future enhancement.



SourceForge Project Page

The project page was not available at the time of this paper's initial publication. Please check http://www.moonlightdesign.org/steve/ for the project page.

Future Directions

Going forward, various aspects of this spell checker framework will likely get improved for better abstractions and increased Document and UI compatibilities. Among some of the improvements are more dictionary-related methods in the SpellChecker interface, server-side SpellCheckerEvents that propagate back to RemoveSpellCheckers, looser couplings between the SpellCheckerContent and the SpellCheckerThread and CustomTextUI classes, a standardized spell checking dialog box, a DelegatedSpellChecker class that splits the SpellChecker delegating functionality out of the CachedSpellChecker, and moving the SpellCheckerContent up to the Document level and renaming it to "SpellCheckerDocument." Aside from these modifications to the spell checker framework, some enhancements to Swing itself can help the framework and possibly other applications. The additions suggested in this document include potential right-click menu standardizations on text fields and other Swing UI objects and a CharacterIterator-returning method on Document and/or Content objects for seamlessly tying into the java.text package. Swing is already highly-extensible and thus does not require any changes to work with spell-as-you-type spell checking, though these changes can potentially make it integrate better.

This project will likely become publicly available on sourceforge.net or an alternative host shortly after this paper's submission and, over the next several months, many of the spell checker framework improvements mentioned will likely get implemented.