


ADOBE® CREATIVE SUITE® 3

ADOBE LINGUISTIC LIBRARY PLUG-INS PROGRAMMING GUIDE





© 2007 Adobe Systems Incorporated. All rights reserved.

Adobe Linguistic Library Plug-ins Programming Guide

Adobe, the Adobe logo, Acrobat, Creative Suite, InDesign, Illustrator, LiveCycle, Photoshop, and Reader are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries. Microsoft and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Mac OS is a trademark of Apple Computer, Inc., registered in the United States and other countries. All other trademarks are the property of their respective owners.

The information in this document is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in this document. The software described in this document is furnished under license and may only be used or copied in accordance with the terms of such license.



Contents

Introduction	5
Terminology	5
Linguistic Library design at a glance	6
Installing Linguistic Library and plug-ins	8
Exercising Adobe Linguistic Library	9
Architecture	10
Plug-in interface specification	10
Plug-in APIs.	11
Run-time architecture	13
Working with Linguistic Library plug-in SDK samples	18
Content of SDK samples.	19
Plug-in interface implementations	20
Compile and test SampleProvider	21
Build your first Linguistic Library plug-in using SampleProvider as a template	22
Linguistic Library plug-in API reference	25
LinguisticPlugIn.h.	25
ILiloProviderPlugIn interface	26
ILiloSpellChecker interface	27
ILiloHyphenation interface	30

Adobe Linguistic Library Plug-ins Programming Guide

Introduction

This programming guide will help you develop plug-ins for Adobe® Linguistic Library. Linguistic Library makes it possible to add linguistic services like spell checkers and hyphenation services to Adobe products, through the Linguistic Library API.

Linguistic Library is not a shrink-wrapped product. Instead, it provides common linguistic services for many Adobe products; thus, it is an important component of these products.

Linguistic Library has an extensible architecture. You can extend the functionality of Linguistic Library to provide customized linguistic services by implementing plug-ins for it. The advantage of implementing plug-ins through Linguistic Library is that these additional linguistic services are available to all Adobe products automatically if they use Linguistic Library. To provide services through individual product extensions, you would have to write programs against every product API.

The intended audience for this document is developers with a basic understanding of Microsoft Component Object Model (COM) technology and Apple's CFPlug-in API (since the Linguistic Library plug-in architecture is based on these technologies).

Terminology

The following terms are used in this document:

- *Client application* — An Adobe application that uses linguistics services provided by Linguistic Library.
- *Lilo* — Shorthand for Adobe Linguistic Library, used in some interface names and folder names.
- *Plug-in* — Except where explicitly specified, “plug-ins” in this document refers to Linguistic Library plug-ins that extend Linguistic Library and provide additional, customized linguistic services.
- *<Plug-in folder>* — The location of the Linguistic Library plug-ins folder. On Mac OS®, this is /Library/Application Support/Adobe/Linguistics/Providers/Plugins. On Windows®, this is C:\Program Files\Common Files\Adobe\Linguistics\Providers\Plugins\.
- *<SDK>* — The path where you installed the Adobe Linguistics Library plug-in SDK.

- *Shell window* — The command-line window. On Mac OS, use the Terminal utility (located in /Applications/Utilities). On Windows, use Command Prompt (located in the Accessories folder from the Start menu).

Linguistic Library design at a glance

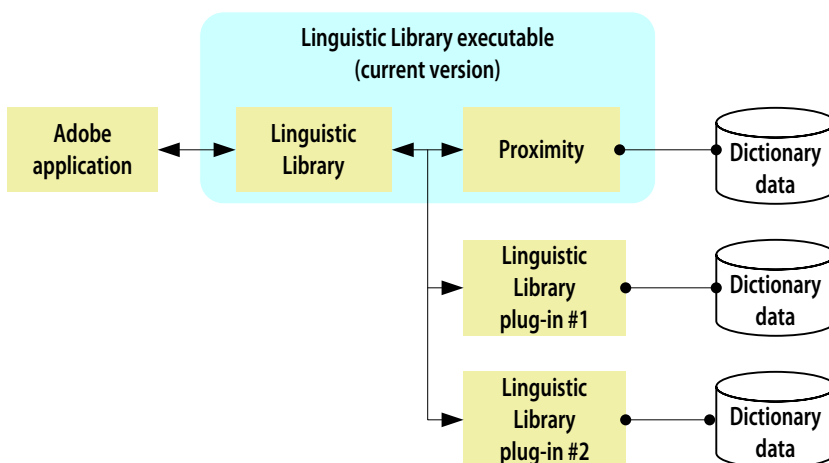
Linguistic Library is a shared library that provides a uniform API to Adobe applications that need linguistic services like spelling, hyphenation, user dictionary, and thesaurus. These client applications include Adobe Acrobat® 8, LiveCycle® Form Designer, Adobe InDesign® CS3, Adobe Illustrator® CS3, and Adobe Photoshop® CS3.

The latest Linguistic Library, shipped with Adobe Creative Suite® 3 applications, is version 3.1. The filename of the library is AdobeLinguistics.framework on Mac OS and AdobeLinguistics.dll on Windows. They are located in the same directory of the application executables and are installed by application installers.

Linguistic Library behaves as an intermediary between Adobe applications and linguistic service providers. On the one hand, it provides a uniform API to Adobe applications, so changes and additions to a linguistic service do not propagate to the applications. On the other hand, it provides an open architecture that allows third-party linguistic-service providers to supply linguistic services through Linguistic Library, so the services can be used by all Adobe applications. The independence achieved by Linguistic Library makes it possible for third-party providers to add new linguistic services even after the Adobe applications are installed.

Figure 1 illustrates the relationships between Adobe applications, Linguistic Library, and linguistic-service providers (through Linguistic Library plug-ins).

FIGURE 1 Relationships between applications, Linguistic Library, and plug-ins



The interface between Adobe applications and Linguistic Library is intended for Adobe internal use and not covered in this document.

The interface between Linguistic Library and linguistic-service providers is the plug-in architecture. COM was chosen for the plug-in specification, and we recommend CFPlug-in for writing plug-ins on Mac OS. For details about the Linguistic Library plug-in architecture, see “Architecture” on page 10. Note that in Linguistic Library version 3.1, Proximity is distributed together with Linguistic Library.

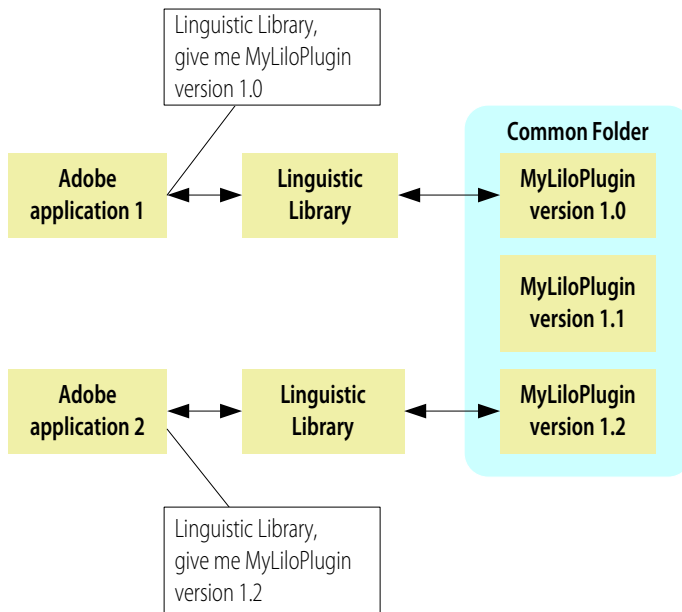
Every Linguistic Library plug-in should have its own dictionary data, which can specify the languages and service it supports. For example, if a custom Linguistic Library plug-in supports US English and provides a spell-checking service, the end user of all Adobe application (like InDesign) can choose the custom spell-checking service in US English.

NOTE: Although Linguistic Library is designed to support all linguistic services, the version 3.1 SDK provides interfaces only for spelling and hyphenation services. Other linguistic services (like a user dictionary) will be added in future releases.

NOTE: While Linguistic Library is accessible by multiple Adobe applications, only InDesign CS3 can use the Linguistic Library plug-ins and choose the spelling and hyphenation service providers via its user interface. Other Adobe applications, like Illustrator and Photoshop, are expected to be able to access these plug-ins in the next release.

Another benefit of Linguistic Library is that it supports plug-in versioning. Plug-ins are installed under version-specific folders and can co-exist with each other. An individual application can select the specific Linguistic Library plug-in version numbers it wants to use. For example, Application A might want to use MyLiloPlug-in Version 1.0, while Application B might want to use version 1.2. See Figure 2.

FIGURE 2 Linguistic Library plug-in versioning



Installing Linguistic Library and plug-ins

Linguistic Library is a shared library that is installed in the same location as the application executables (Windows) or linked into the application (Mac OS) during Adobe application installation. Linguistic Library’s own data file and any Linguistic Library plug-ins that ship with Adobe applications also are installed during the application-installation process, but to the common Linguistic Library data and plug-in folder. See [Table 1](#) for the default location of these files.

TABLE 1 *Linguistic Library installation locations*

Component	Windows	Mac OS
Shared library	AdobeLinguistics.dll at the same location as the application. For example, C:\Program Files\Adobe\Adobe InDesign CS3\AdobeLinguistic.dll.	In the Frameworks folder inside the application’s package contents. For example, /Applications/Adobe Illustrator CS3/Adobe Illustrator.app/Contents/Frameworks/AdobeLinguistic.framework/Versions/3/AdobeLinguistic.
Data file	Common Program Files. For example, C:\Program Files\Common Files\Adobe\Linguistics\LanguageNames\DisplayLanguageNames.en_US.txt and C:\Program Files\Common Files\Adobe\Linguistics\Providers\Proximity11\brt04.lex.	Common application-support folder. For example, /Library/Application Support/Adobe/Linguistics/LanguageNames/DisplayLanguageNames.en_US.txt and /Library/Application Support/Adobe/Linguistics/Providers/Proximity11/brt04.lex.
Plug-ins	Common Linguistic Library plug-ins folder. For example, C:\Program Files\Common Files\Adobe\Linguistics\Providers\Plugins\WRLiloPlug-in1.0\.	Common Linguistic Library plug-ins folder. For example /Library/Application Support/Adobe/Linguistics/Providers/Plugins/WRLiloPlug-in1.0/.

NOTE: (Windows only) To install custom Linguistic Library plug-ins after an application is installed, you need to register with the systems in addition to copying plug-ins to the specified folder. You should login as a user with administrator privileges, open a command console, and execute `regsvr32 <plug-in DLL name>`. (If you use Windows Vista, you need start the command console with administrator privileges by clicking the Start menu, then right clicking on the command console and choosing “Run as administrator” menu).

To unregister the DLL, run the command `regsvr32 /u <plug-in DLL name>`.

NOTE: Acrobat and Acrobat Reader® install Linguistic Library to different locations.

Exercising Adobe Linguistic Library

Linguistic Library does not have a user interface; to access the functionality and services provided by Linguistic Library and its plug-ins, you must run an Adobe application. (In CS3, only InDesign has the appropriate user interface. Other Adobe applications, like Illustrator and Photoshop, are expected to be able to access Linguistic Library plug-ins and services in the next release.) In InDesign CS3, to check out the spelling service provided by Linguistic Library, follow these steps:"

1. Install InDesign CS3. (You may install one of the Adobe CS3 suites that include InDesign.)
2. Create one or more custom Linguistic Library plug-ins. See [“Build your first Linguistic Library plug-in using SampleProvider as a template” on page 22.](#)
3. Install custom Linguistic Library plug-ins. See [“Installing Linguistic Library and plug-ins” on page 8.](#)
4. Launch InDesign.
5. (Optional) If you want the preference set in the next step to apply to only one document, open an existing document or create a new one.
6. Bring up the dictionary preference dialog, by choosing Edit > Preference > Dictionary... (Windows) or InDesign > Preference > Dictionary... (Mac OS).
7. Select a language. Linguistic Library 3.1 supports more than 40 languages. If your custom Linguistic Library plug-in supports another language, that language appears in the drop-down list. Multiple providers can support the same language; for example, your custom Linguistic Library plug-in also may support US English.
8. Choose the spelling or hyphenation service provider for the selected language. If your custom Linguistic Library plug-in provides a service for the selected language, the service appears in the drop-down list for spelling or hyphenation. If you choose Adobe-supported languages and services, you are exercising Adobe-provided Linguistic Library functions; if you choose a service provided by your own, custom, Linguistic Library plug-in, you are exercising your custom plug-in.
9. Close the dictionary preference dialog.
10. If you have not already done so, create a new document. Bring the Character panel to the front, and make sure your chosen language is shown at the bottom of the panel.
11. Create a text frame, and type a few words.
12. To test spelling, bring up the spelling dialog by choosing Edit > Spelling > Check Spelling... The dialog steps through the text with the spelling service you chose previously. Make sure it flags words that are not in the dictionary.
13. To test hyphenation, resize the text frame to see whether hyphenation works as expected.

Architecture

The Linguistic Library plug-in architecture is designed for third-party vendors to provide their spelling and other linguistic services to Adobe applications through Linguistic Library's API. The plug-in architecture comprises the following:

- Plug-in interface specification
- Plug-in APIs
- Run-time architecture

These components are discussed in the following sections.

Plug-in interface specification

One design goal of the plug-in interface specification is to make it as compatible as possible at the source level on both Windows and Mac OS.

On Windows, the standard software component architecture is COM, which defines the binary interface of a component for interoperability. COM also has guidelines for writing component software, to facilitate interface queries and dynamic linking at run time. COM has existed on Windows since 1995. It is well documented and supported fully by the Windows run-time environment.

COM offers many features, but in Linguistic Library's architecture, we use only rudimentary features like the IUnknown interface, its usage protocol, and the component-registry functions to find and load Linguistic Library plug-ins.

On Mac OS, the Core Foundation library provides a plug-in architecture known as CFPlug-in that is derived from COM's IUnknown interface; therefore, it was natural to adopt COM as the plug-in binary interface on both platforms. This standardizes the calling convention at the source-code level and eliminates the need for another library to provide the plug-in architecture on Mac OS. The CFPlug-in is in the native Mach-o format, and Mac OS provides APIs for loading and basic run-time support.

Component registration on Windows differs from Mac OS, but Linguistic Library takes care of component management for applications and plug-ins. For details on the registration behavior, see [“Plug-in registration” on page 15](#).

GUID in COM

An important aspect of COM is that it uses GUID (Globally Unique Identifier) for its class ID and interface ID. Each COM object contains exactly one class ID, or CLSID, and one interface ID (IID) for each interface it supports. For example, a Linguistic Library plug-in supporting the provider and the spelling interface has three GUIDs: one CLSID and two IIDs. Usually, the CLSID and one IID are used together to locate a COM object that supports an interface.

On Mac OS, UUID (Universally Unique Identifier) is used, although it is identical to the GUID. Instead of a CLSID, the CFPlug-in has a Type ID, and the IID is the same as on Windows.

IUnknown

At its most basic level, the COM interface specification is a table of function pointers that resembles the vtable of an abstract class in C++. The root class that all COM classes must inherit from is the IUnknown class, defined as follows:

```
interface IUnknown
{
    HRESULT QueryInterface(
        REFIID iid,
        void ** ppvObject) = 0;
    ULONG AddRef(void) = 0;
    ULONG Release(void) = 0;
};
```

On Mac OS, the IUnknown interface is defined as follows:

```
interface IUnknown
{
    virtual HRESULT __stdcall QueryInterface(
        const IID& iid
        void **ppv) = 0;
    virtual ULONG __stdcall AddRef() = 0;
    virtual ULONG __stdcall Release() = 0;
};
```

The corresponding C struct is defined as follows:

```
#define IUNKNOWN_C_GUTS
void *_reserved;
HRESULT (STDMETHODCALLTYPE *QueryInterface)
(void *thisPointer, REFIID iid, LPVOID *ppv);
ULONG (STDMETHODCALLTYPE *AddRef)(void *thisPointer);
ULONG (STDMETHODCALLTYPE *Release)(void *thisPointer);
```

The C struct could be used to define the function-pointer table for IUnknown and any other interface derived from IUnknown, if the compiler does not support COM interface generation. An example is listed below:

```
typedef struct TestInterfaceStruct {
    IUNKNOWN_C_GUTS;
    void (*fooMe)( void *this, Boolean flag );
} TestInterfaceStruct;
```

As the C struct demonstrates, the COM interface specification basically is a simple function-pointer table. As long as the plug-in follows the COM protocols for the IUnknown interface, the object behavior is standardized.

Plug-in APIs

Linguistic Library plug-in APIs are shipped with the SDK and located in <SDK>\api\Plugin. The plug-in API is grouped into the following categories:

- Provider
- Spelling
- Hyphenation

You must implement the provider API, since Linguistic Library uses it to query the plug-in on what feature it supports, such as services and languages supported. The other categories are optional. For example, if you have only spelling technology, you would implement the spelling API in your plug-in.

In the future, new categories will be created and new APIs will be published. For example, user dictionary and thesaurus services and APIs may be added in a future release of the Linguistic Library plug-in SDK. Since the API set is based on the COM interface, it is not statically bonded and can be extended without requiring existing Linguistic Library clients to recompile. A new plug-in could support a new API yet still serve existing Linguistic Library clients, if it continues to support the existing interface.

Memory-allocation policy

Whenever the function calls for a plug-in to return a word or an array of data, the plug-in needs to allocate the memory with the SysMemAllocate() function, which is a COM memory-allocation function that ensures the safe allocation of memory. Linguistic Library is responsible for disposing of the memory allocated by the plug-in.

If Linguistic Library allocates an object and passes it to the plug-in, Linguistic Library still is responsible for de-allocating it. The plug-in should not de-allocate these objects itself.

COM data types

The Linguistic Library plug-in API uses COM data types like BSTR, SHORT, LONG, and CHAR. They are defined in the public COM documentation, available on the Microsoft Web site.

Error code

When a function succeeds, it returns S_OK. When a function fails, a plug-in is required to return a COM-style error code, like E_OUTOFMEMORY. For appropriate errors, see the COM error-code types.

Interfaces

Table 2 summarizes supported interfaces in Linguistic Library version 3.1. For detailed descriptions of the interfaces, including every function of each interface, see “Linguistic Library plug-in API reference” on page 25.

TABLE 2 Interface summary

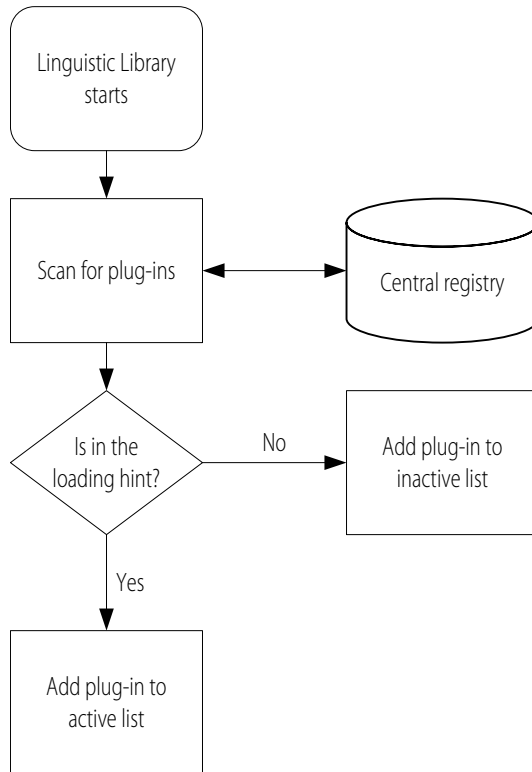
Interface	Description
ILiloHyphenation	Provides functions for hyphenation services: gets possible hyphenation points and splits a word at a hyphenation point.
ILiloProviderPlugIn	Tells Linguistic Library what services it provides, what potential languages it supports for a specified service, and its name in a Unicode string. This interface allows Linguistic Library clients to implement user-interface features.

Interface	Description
ILiloSpellChecker	Provides functions for spelling services: checks the spelling of a word and provides a list of suggestions to correct a misspelled word.

Run-time architecture

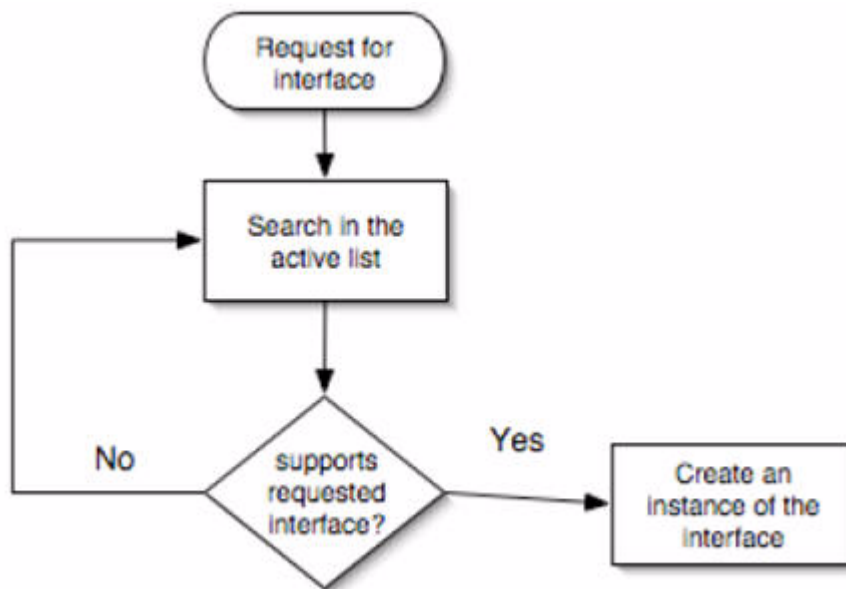
When Linguistic Library is starting up, it scans for plug-ins from a central registry. The central registry is implemented differently on each platform; it is discussed in the “[Plug-in registration](#)” on page 15. For each plug-in found, Linguistic Library checks which interfaces it supports. [Figure 3](#) is a flow chart of the start-up process.

FIGURE 3 Linguistic Library start-up process



The initialization is done primarily by reading information from the central registry instead of scanning and loading each plug-in, to reduce initialization time. No plug-in is loaded at this point.

When a Linguistic Library client requests a linguistic service (like spelling) from a particular provider, Linguistic Library requests an appropriate interface from that provider, or the plug-in. If the plug-in supports the interface, Linguistic Library asks it to create an instance of the interface and gives it back to the Linguistic Library client. [Figure 4](#) illustrates the workflow.

FIGURE 4 Linguistic Library request for interface

After Linguistic Library plug-ins are loaded, Linguistic Library acts as an agent between applications and linguistic services. Linguistic Library plus plug-ins is an encapsulated whole. [Figure 5](#) shows the conceptual relationship between plug-in components.

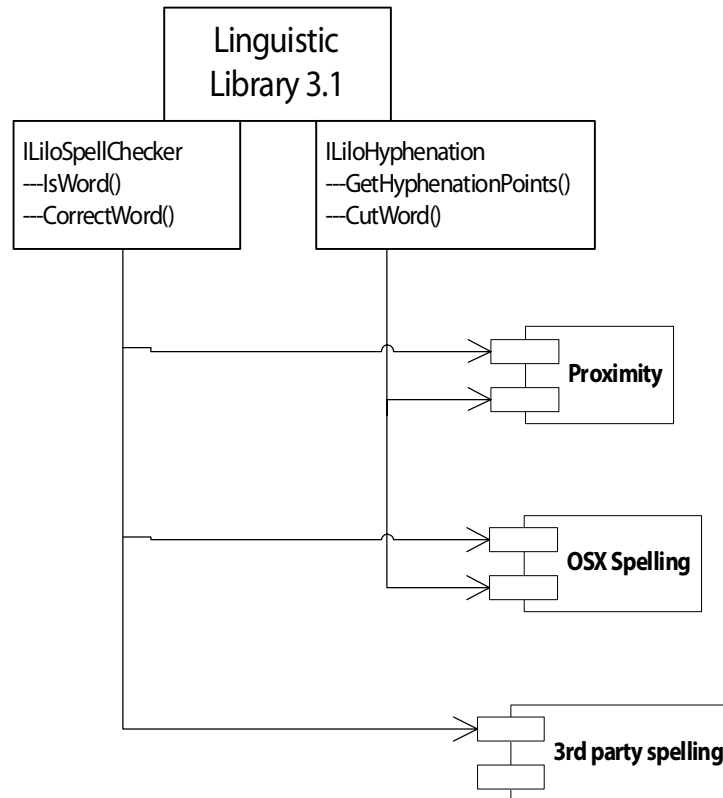
FIGURE 5 Relationships between Linguistic Library and its plug-ins

Figure 5 illustrated two interfaces:

- *ILiloSpellChecker* provides the functions the plug-in needs to support spell checking. For instance, Linguistic Library calls `IsWord()` to check the spelling of a word. If the word is not spelled correctly, Linguistic Library calls `CorrectWord` to get a list of suggestions.
- *ILiloHyphenation* specifies the functions a plug-in needs to implement hyphenation.

The figure shows three plug-ins: Proximity, OSX Spelling, and a 3rd party spelling plug-in. The Proximity and OSX Spelling plug-ins support both the *ILiloSpellChecker* and *ILiloHyphenation* interfaces. In the example in the figure, the third-party plug-in intentionally does not support hyphenation, so it provides an implementation for only the *ILiloSpellChecker* interface.

Plug-in registration

After a Linguistic Library plug-in is copied to its desired location, it needs to register itself so Linguistic Library can find it at run time. The registration mechanism differs on Mac OS and Windows platform, but the concept is similar: Linguistic Library relies on a central registry to search for a plug-in to load. The central registry is discussed in more detail in “[Central registry](#)” on page 18. For Adobe-supplied plug-ins, registration is done as a part of installation.

When a plug-in is installed, several pieces of information are written in the central registry to properly register the plug-in. The most important ones are as follows:

- *Plug-in location* — A plug-in usually is invoked to register itself. On Windows, this is how a COM component registers. During the registration process, the plug-in location, its GUIDs, and other COM-conforming information are written into the Windows registry. On Mac OS, a CFPlug-in can choose between static or dynamic registration. The CFPlug-inDynamicRegistration determines which type of registration is required. A static registration means all the necessary information is stored in its property-information list, or plist. Dynamic registration makes the operating system load the plug-in and call its registration functions. The plist should contain the name of the function that should be called for dynamic registration. For Linguistic Library 3.1 plug-ins, only static registrations are allowed.
- *Plug-in category* — This describes what linguistic services the plug-in supports. Linguistic Library must quickly determine which linguistic services each plug-in supports, without having to load each plug-in at start-up time. Either the installer or the plug-in itself can write this information.
- *Optional plug-in loading hint* — Linguistic Library's client supplies this to indicate any specific plug-in that should be loaded. Plug-ins have no control over loading hints.

On Windows, a plug-in should implement and export two well known DLL entry points, DllRegisterServer and DllUnregisterServer. This allows the standard registry command Regsvr32 to register a plug-in.

On Mac OS, a plug-in carries the attributes listed in [Table 3](#) in its plist, which is stored in the CFBundle.

TABLE 3 Mac OS Plug-in plist attributes

Attribute name	Value	Purpose
CFPlug-inDynamicRegisterFunction	String	Function name to call for dynamic registration.
CFPlug-inDynamicRegistration	Yes, No	Whether dynamic registration is required
CFPlug-inFactories	(factory UUID, string)	Used for static registration. <i>String</i> is the factory function name that creates an instance of the UUID, which represents the interface it supports.
CFPlug-inTypes	(Type UUID, array of factory UUID)	Used for static registration. This defines what interface the plug-in supports. Each factory UUID corresponds to an interface it supports.
CFPlug-inUnloadFunction	String	Function name to call to unload this plug-in.

These attributes defines how a CFPlug-in is registered to its host, but they do not register the category or write anything to the central registry.

Component category

Component category is introduced in the COM specification as a way to categorize components. As the number of COM objects increases, there must be a way to search for components that support a particular interface.

On Windows, the registry is used to store this category information. When a COM object is loaded—or in Window's terminology, activated—it registers itself in the registry and declares which interfaces it supports. The Component Category Manager is provided to facilitate the registration and search.

Using Microsoft OLE/COM viewer, many component categories are defined in the Windows registry. Some examples are .Net Category, Document Objects, and Java Classes. Linguistic Library searches the Adobe Linguistic Provider category.

For example, when a Linguistic Library plug-in is installed, it registers itself and declares that it supports the IProvider and ISpellChecker interfaces. It adds registry entries to the Adobe Linguistic Provider category. Later, Linguistic Library starts up and asks the Component Category Manager to search this category for a list of COM objects that support the IProvider and ISpellChecker interfaces, and it receives the CLSID and the IIDs of the Linguistic Library plug-in. Linguistic Library activates the plug-in and queries for its supported features.

On Mac OS, there is no service comparable to the Windows registry to register a plug-in for a specific category; therefore, Linguistic Library scans the Linguistic Library plug-in folder to find plug-ins. If a plug-in is not in this folder, it is not loaded.

Make sure Mac OS plug-ins are installed at a central location under the /Library/Application Support/Adobe/Linguistics/Providers/Plugins folder.

Plug-in object creation

How does a plug-in get loaded, and how does the object get created for a requested interface? Generally, a plug-in needs to support a factory function that knows how to create an instance of an interface. Linguistic Library uses this factory function to create objects for the interfaces in which it is interested. The implementation details, however, differ on Windows and Mac OS.

On Windows, the Component Category Manager returns a CLSID and an IID for a specific interface that Linguistic Library is looking for. Linguistic Library uses the Windows Service Control Manager, or SCUM, to locate and load the plug-in. SCUM locates the registry entry associated with the CLSID, which usually is the InprocServer32 subkey, finds its DLL location, and loads it. When the plug-in is loaded, the COM manager asks the plug-in for its factory function, which in turn returns the IClassFactory interface. IClassFactory is a standard interface defined in COM specifically for creating instances for a given interface. Linguistic Library gets an instance of IClassFactory from the plug-in, which knows how to create objects for the interfaces it supports. For example, Linguistic Library will ask the IClassFactory to create an instance of the ILiloProviderPlugIn interface, then it can use it to query ILiloSpellChecker to do a spelling check. If Linguistic Library asks for an interface that is not supported by the plug-in, its IClassFactory returns NULL.

To return the `IClassFactory` interface, the plug-in should implement and export the `DllGetClassObject` entry point, which returns an instance of `IClassFactory`.

On Mac OS, Linguistic Library needs to do the work of the Component Category Manager in locating a plug-in that supports the requested interface. Once one is located, Linguistic Library uses `CFPlug-in` API to get its factory function. This function behaves like the `IClassFactory` on the Windows platform: it creates an instance of the interface that Linguistic Library requests. If the interface is not supported, `NULL` is returned. A `CFPlug-in` component usually identifies its factory function in its `plist`.

Plug-in lifecycle

The COM interface specification requires each interface to implement a reference count. COM components on Windows also are encouraged to implement a total object count, so when all objects are released, the plug-in can be unloaded by Windows at an appropriate time. On Windows, the plug-in needs to implement and export the `DllCanUnloadNow` entry point. If `DllCanUnloadNow` returns `true`, the plug-in is unloaded.

On Mac OS, the plug-in provides a function to unload, but it is up to the host to decide whether to unload the plug-in. In this case, Linguistic Library does not unload the plug-in until the Linguistic Library client is ready to quit.

Central registry

The plug-in run-time environment relies on a central registry to find plug-ins and their attributes, like categories. On the Windows platform, there is the Windows registry that the COM components use to register themselves. Linguistic Library also uses the Component Category Manager to find the linguistic categories defined by Linguistic Library. The Component Category Manager talks to the Windows registry to manage the component categories.

On Mac OS, there is no central registry like the Windows registry; instead, Linguistic Library looks for plug-ins in the `Plugin` folder at start-up time. It looks for the component-category information stored in each plug-in's `plist` and stores the category information in a memory cache. Although plug-ins with static registration are not loaded, reading the `plist` of each plug-in might take awhile.

Working with Linguistic Library plug-in SDK samples

This section provides guidance on how to work with the interfaces and sample code provided in the SDK.

Content of SDK samples

Enclosed with this SDK, you will find both Windows and Mac OS sample linguistic-service providers, with projects and source code:

- The `<SDK>/api/` folder contains public-interface header files.
- The `<SDK>/build/XCode` folder contains a Mac OS sample project and related files, including `info.plist`.
- The `<SDK>/build/VC.Net2005` folder contains a Windows Visual Studio sample project file.
- The `<SDK>/sdksamples` folder contains source files. It contains open source, Macintosh-only, common files in the `common/mac` folder. The `SampleProvider` folder has platform-independent code implementing public interfaces, as well as platform-specific files under the “mac” and “win” subfolders.

API files

There are five public header files:

- `ILiloProviderPlugin.h`, `ILiloSpellChecker.h`, and `ILiloHyphenation.h` are in the `<SDK>/api/Plugin` folder. They are the interfaces with which a plug-in is programmed.
- `LinguisticConditions.h` and `LinguisticPlugin.h` are in the `<SDK>/api/includes` folder. They define common variables used in cross-platform implementation.

Since these interfaces define the contract between Linguistic Library and a Linguistic Library plug-in, they should not be changed.

Cross-platform source files

Open either `SampleProvider.vcproj` on Windows or `SampleProvider.xcodeproj` on Mac OS. You will find the following code in both projects:

- `PlugInProvider.cpp` and `PlugInProvider.h` implement the `ILiloProviderPlugin` interface.
- `PlugInSpellChecker.cpp` and `PlugInSpellChecker.h` implement the `ILiloSpellChecker` interface.
- `PlugInHyphenation.cpp` and `PlugInHyphenation.h` implement the `ILiloHyphenation` interface.

These are core implementation files that are located in the `<SDK>/sdksamples/sampleprovider` folder.

Windows-specific source files

In addition to common files, the following files are included in `SampleProvider.vcproj`:

- `SampleProvider.def` — A module-definition file that defines the exports of the plug-in DLL.
- `SampleProvider.cpp` and `SampleProvider.h` — The main DLL source file that implements exported DLL entries like `DllRegisterServer` and `DllUnRegisterServer`.

These files are located in the `<SDK>/sdksamples/sampleprovider/win` folder.

Mac OS-specific source files

On Mac OS, Xcode projects include files that provide the CoreFoundation equivalent of Windows COM and Automation:

- Guiddef.h, initguid.h, objbase.h, oleauto.cpp, oleauto.h, and winbase.h — These are open-source files. They are located in the <SDK>/sdksamples/common/mac folder.
- Server.cpp and Server.h — These are “glue code” that implement COM on Mac OS. They are located in the <SDK>/sdksamples/sampleprovider/mac folder.

Plug-in interface implementations

ILiloProviderPlugin

The first interface you need to implement is `ILiloProviderPlugin`. This is the required interface that tells Linguistic Library that the plug-in provides a certain service.

`PlugInProvider.h` and `PlugInProvider.cpp` implement two classes:

- `PlugInProviderFactory` — Called to create `PlugInProvider` instance when the plug-in is loaded. On Windows, it is called within the `DllGetClassObject` function (`SampleProvier.cpp`). On Mac OS, the plug-in entry point is `PluginFactory`, which creates a `PlugInProviderFactory` class to instantiate `PlugInProvider` via `QueryInterface`.
- `PlugInProvider` — A COM object that inherits from all three interfaces (`LiloProviderPlugin`, `ILiloSpellChecker`, and `ILiloHyphenation`). It creates `PlugInSpellChecker` and `PlugInHyphenation` in its constructor and keeps references to them throughout their lifetime. Function calls to `PlugInSpellChecker` and `PlugInHyphenation` are simply forwarded to them.

Methods on `ILiloProviderPlugin` interface are implemented in `PlugInProvider` directly. They are as follows:

- `GetProviderName()` — Returns the name of the linguistics-service provider.
- `GetServicesSupported()` — Sets up an array of services provided, like `kLMSpellingService` and `kLMHyphenationService`.
- `GetLanguagesSupported()` — Specifies the languages of a specific supported service.

NOTE: You can have your own, customized implementation of the `PlugInProvider` class.

ILiloSpellChecker

`PlugInSpellChecker.cpp` and `PlugInSpellChecker.h` implement the `ILiloSpellChecker` interface. In addition to `IUnknown` methods, `PlugInSpellChecker` implements the following:

- `GetLanguageRef()` — Returns a language reference.
- `IsWord()` — Returns whether a word is included in the dictionary.
- `CorrectWord()` — Gives a list of suggestions for correcting a misspelled word.
- `GetPropertyList()`, `GetProperty()`, and `SetProperty()` — Manipulate spell-checking options.

PlugInSpellChecker defines private-member valuables that support these methods. For example, `m_SupportedLanguages` stores a list of languages supported, and `m_Capability` stores a list of spell-checker options. These language-definition and spell-checker options are defined in the `LinguisticPlugIn.h` header file.

Only two words (“Adobe” and “Lilo”) are included in the library; they are hard-coded in the `PlugInSpellChecker` constructor, as shown below:

```
PlugInSpellChecker::PlugInSpellChecker()
{
    // initialize my dictionary
    m_dictionary.push_back(SysAllocString(OLESTR("Adobe")));
    m_dictionary.push_back(SysAllocString(OLESTR("Lilo")));
    ...
}
```

ILiloHyphenation

`PluginHyphenation.cpp` and `PluginHyphenation.h` implement the `ILiloHyphenation` interface. In addition to `IUnknown` methods, `PluginHyphenation` implements the following:

- `GetHyphenLanguageRef()` — Returns a language reference.
- `GetHyphenationPoints()` — Returns a list of positions where a word can or cannot be broken up.
- `CutWord()` — Cuts a word into two parts.

As a reference implementation, the sample has hard-coded hyphenation points for only one word, “laboratory”; see below. You should design your own data structures for real-world hyphenation.

```
PlugInHyphenation::GetHyphenationPoints()
STDMETHODIMP PlugInHyphenation::GetHyphenationPoints(...)
{
    ...
    BSTR gMyWord = SysAllocString(OLESTR("laboratory"));
    ...
}
```

Compile and test SampleProvider

To run the sample code on Windows, follow these steps:

1. Use Microsoft Visual C++ 2005 to open `<SDK>\build\VC.Net2005\SampleProvider.vcproj`.
2. Build the project. You may notice there are two C4267 warnings, complaining of `size_t` to `SHORT` and `int` conversion; ignore these warnings.
3. The compiled plug-in named `SampleProvider.dll` is located in `<SDK>\build\VC.Net2005\debug` or `<SDK>\build\VC.Net2005\release`.
4. Follow the instructions in [“Installing Linguistic Library and plug-ins” on page 8](#) and [“Exercising Adobe Linguistic Library” on page 9](#) to install a custom Linguistic Library plug-in

and exercise Linguistic Library. Do not forget the step that executes the “regsvr32 SampleProvider.dll” command, to register with the operating system.

To run the sample code on Mac OS, follow these steps:

1. Use Apple Xcode 2.4.1 to open `<SDK>/build/XCode/SampleProvider.xcodeproj`.
2. Build the project.
3. The compiled plug-in named `SampleProvider.bundle` is located in `<SDK>/build/XCode//build/debug` or `<SDK>/build/XCode/build/release`. Follow the steps in [“Installing Linguistic Library and plug-ins” on page 8](#) and [“Exercising Adobe Linguistic Library” on page 9](#) to use the `SampleProvider` plug-in.

If InDesign is running, restart it. In the Preferences > Dictionary Dialog, you should now see Hyphenation and Spelling options for the sample provider. During your testing, you should notice that all words other than “Adobe” and “Lilo” are flagged as misspelled. For hyphenation, “laboratory” may be hyphenated in position 3 or 5 (after “b” or the first “r”); all other words may be hyphenated at position 2. These are due to our hard-coded spelling and hyphenation implementations.

Build your first Linguistic Library plug-in using SampleProvider as a template

`SampleProvider` is the starting point for your first plug-in. To build your first plug-in based on `SampleProvider`, use the steps in the following sections.

Method 1: Copy and modify SampleProvider

You can build your plug-in by copying and modifying the existing `SampleProvider`:

1. Copy the entire `<SDK>` folder to a new location. If you use a source-code control system to track your code, this could be your initial version.
2. Choose a new project name for the plug-in. You may not want your plug-in to use the same name as `SampleProvider`. Here, we assume you name your plug-in `MyProvider`.
3. Modify the Windows project:
 - Open the `<SDK>\build\VC.Net2005` folder, and rename `SampleProvider.vcproj` as `MyProvider.vcproj`.
 - Open `MyProvider.vcproj` as plain text, and replace all occurrences of “`SampleProvider`” with “`MyProvider`” and “`sampleprovider`” with “`myprovider`” (case sensitive).
4. Change source files:
 - Open `<SDK>sdksamples` and rename the `sampleprovider` folder to be `myprovider`.
 - Open the `<SDK>sdksamples\myprovider\win` folder, and rename `SampleProvider.cpp`, `SampleProvider.h`, and `SampleProvider.def` as `MyProvider.cpp`, `MyProvider.h`, and `MyPro-`

vider.def. (This step is not necessary if you did not change the filename included in the vcproj in the previous step).

- In the `<SDK>\sdksamples\myprovider` folder, replace all occurrences of “SampleProvider” with “MyProvider” and “sampleprovider” with “myprovider” (case sensitive), especially the ones in `MyProvider.cpp`, `MyProvider.h`, and `MyProvider.def`.
5. Modify Mac OS project files accordingly:
 - Open the `<SDK>/build/XCode` folder, and rename `SampleProvider.xcodeproj` as `MyProvider.xcodeproj`.
 - Rename `SampleProvider_English.lproj` as `MyProvider_English.lproj`. Rename `SampleProvider_Info.plist` as `MyProvider_Info.plist`.
 - Open `MyProvider.xcodeproj/project.pbxproj` as plain text, and replace all occurrences of “SampleProvider” with “MyProvider” and “sampleprovider” with “myprovider” (case sensitive).
 - Open `<SDK>/build/XCode/MyProvider_Info.plist`, and replace `SampleProvider` with `MyProvider` in the `<key>CFBundleIdentifier</key>` section.
 6. Assign a name to your provider. Do not use the same provider name as `SampleProvider` (“Adobe Linguistic Sample Plug-in”). Open `PlugInProvider.cpp`, and change the implementation of `GetProviderName(BSTR* outname)` to set a new provider name (like “My Provider Plug-in”).
 7. (Windows only) Make your plug-in loadable on Windows:
 - From the command shell, run `guidgen` to generate a new GUID.
 - Open `MyProvider.cpp`, and replace the `DEFINE_GUID` statement with a new GUID value, to define a new CLSID.
 - Find the `gRegTable` definition, and replace all CLSID strings with the same new GUID value.
- NOTE:** Do not change the GUID of interface IDs defined in files under the headers folder.
8. (Mac OS only) Make your plug-in loadable on Mac OS. Linguistic Library uses a different mechanism to load a plug-in on Mac OS. In contrast to Windows, *do not* change the GUID of `CLSID_LiloProviderPlguIn` as defined in `Server.cpp`.
 9. Change the spelling and hyphenation services to your own implementations. You can freely change `PlugInSpellChecker.cpp` and `PlugInHyphenation.cpp`, as long as the methods on their corresponding interfaces are implemented. You can even add new support classes, CPP files, data files, etc. into the project. You also can alter the `PlugInProvider` implementation as needed.
 10. Test your own plug-in. Follow the same steps as for testing `SampleProvider`. You should see both “Adobe Linguistic Sample Plug-in” and your own provider name (“My Provider Plug-in”) as choices for the end user.

Method 2: Add a new provider

Alternately, you can add a new provider by duplicating and then renaming Mac OS and Windows projects as well as `<SDK>\sdksamples\sampleprovider` at their current locations, following steps similar to those above to create your own plug-in:

1. Choose a new project name for your plug-in. Here, we assume you name your plug-in `MyNextProvider`.
2. On Windows, open the `<SDK>\build\VC.Net2005` folder, copy `SampleProvider.vcproj`, and rename it as `MyNextProvider.vcproj`.
3. Open `MyNextProvider.vcproj` as plain text, and replace all occurrences of “SampleProvider” with “MyNextProvider” and “sampleprovider” with “mynextprovider” (case sensitive).
4. Open the `<SDK>/build/XCode` folder and rename related files. Rename `SampleProvider.xcodeproj` as `MyNextProvider.xcodeproj`, `SampleProvider_English.lproj` as `MyNextProvider_English.lproj`, and `SampleProvider_Info.plist` as `MyNextProvider_Info.plist`.
5. Open `MyNextProvider.xcodeproj/project.pbxproj` as plain text, and replace all occurrences of “SampleProvider” with “MyNextProvider” and “sampleprovider” with “mynextprovider” (case sensitive).
6. Open `<SDK>/build/XCode/MyProvider_Info.plist`, and replace `sampleprovider` with `mynextprovider` in the `<key>CFBundleIdentifier</key>` section.
7. Open the `<SDK>sdksamples` folder, copy the `sampleprovider` folder, and rename it as `mynextprovider`.
8. Open the `<SDK>sdksamples\mynextprovider\win` folder, and rename `SampleProvider.cpp`, `SampleProvider.h`, and `SampleProvider.def` as `MyNextProvider.cpp`, `MyNextProvider.h`, and `MyNextProvider.def`. (This step is not necessary if you did not change the filename included in the `vcproj` in the previous step).
9. Replace all instances of “SampleProvider” in files in the `<SDK>\sdksamples\myprovider` folder. (Only the ones in `MyNextProvider.cpp`, `MyNextProvider.h`, and `MyNextProvider.def` matter for this sample).
10. Choose a new name for your provider. Do not use the same provider name as existing ones. Open `PlugInProvider.cpp`, and change the implementation of `GetProviderName(BSTR* outname)` to set a new provider name (like “My Next Sample Plug-in”).
11. Make your plug-in loadable on Windows:
 - From the command shell, run `guidgen` to generate a new GUID.
 - Open `MyNextProvider.cpp`, and replace the `DEFINE_GUID` statement with a new GUID value, to define a new CLSID.

- Replace the const char* MyNextProviderCLSID statement with a new GUID value string.
- Find the gRegTable definition, and replace all CLSID strings with the same new GUID value string.

NOTE: Do not change the GUID of interface IDs defined in files under the API folder.

12. Make your plug-in loadable on Mac OS. Linguistic Library uses a different mechanism to load a plug-in on Mac OS. In contrast to Windows, *do not* change the GUID of CLSID_LiloProviderPlguIn as defined in Server.cpp.
13. Change the spelling and hyphenation services to your own implementations. You can freely change PlugInSpellChecker.cpp and PlugInHyphenation.cpp, as long as the methods on their corresponding interface are implemented. You can even add new support classes, CPP files, data files, etc. into the project. You also can alter the PlugInProvider implementation as needed.
14. Test your own plug-in. Follow the same steps as for testing SampleProvider. You should see existing providers like “Adobe Linguistic Sample Plug-in” and “My Provider Plug-in,” as well as your own “My Next Sample Plug-in” as choices for the end user.

Linguistic Library plug-in API reference

This section provides detailed descriptions of Linguistic Library plug-in programming interfaces.

LinguisticPlugIn.h

LinguisticPlugIn.h defines commonly used enums and constants.

Language code

The first segment of the file defines the ISO language code; for example, en_US is American English and fr_FR is French. For a detailed list of language codes, see the source code.

Linguistic service

We defined four linguistic services as enums: kLMSpellingService, kLMHyphenationService, kLMThesaurusService, and kLMUserDictionaryService. Adobe Linguistic Library version 3.1 supports only spelling and hyphenation services.

Property type

LM_PlugInPropertyType defines the property types that can be set in a plug-in. It is used to determine the capability of a plug-in. These are used as spell-checker options.

Hyphenation-point preference

There are three types of hyphenation-point preference: kLMHyphenNormalPoint, kLMHyphenPreferredPoint, and kLMHyphenNonpreferredPoint. They are used in Linguistic Library hyphenation implementations to specify how to split a word.

ILiloProviderPlugIn interface

ILiloProviderPlugIn tells Linguistic Library what services it provides, what potential languages it supports for a specified service, and its name in a Unicode string. This interface allows Linguistic Library clients to implement user-interface features. The interface is defined as follows:

```
class ILiloProviderPlugIn : public IUnknown
{
public:
virtual /* [helpstring] */
HRESULT STDMETHODCALLTYPE GetProviderName(
/* [retval][out] */ BSTR *outName
) PURE;

virtual /* [helpstring] */
HRESULT STDMETHODCALLTYPE GetServicesSupported(
/* [out] */ SHORT *outCount,
/* [out] */ SHORT **outArray // array of LM_ServiceType
) PURE;

virtual /* [helpstring] */
HRESULT STDMETHODCALLTYPE GetLanguagesSupported(
/* [in] */ SHORT inServiceType,
/* [out] */ SHORT *outCount,
/* [out] */ char** *outArray // array of LM_LanguageCode
) PURE;
};
```

GetProviderName

This function allows Linguistic Library to get the name of a plug-in. The string is a Unicode string, and spaces are allowed. The maximum length is 64 characters.

Syntax:

```
HRESULT GetProviderName ( BSTR *outName );
```

TABLE 4 *GetProviderName parameter table*

Name	Use	Description
outName	[out]	The name of the plug-in.

GetServicesSupported

This function allows Linguistic Library to get a list of services from the plug-in. Services are defined as the LM_ServiceType, which include kLMSpellingService and kLMHyphenationService.

Syntax:

```
HRESULT GetServicesSupported ( SHORT *outCount, SHORT **outArray );
```

TABLE 5 *GetServicesSupported parameter table*

Name	Use	Description
outCount	[out]	A count of the services.
outArray	[out]	An array of services for Linguistic Library to pick up.

GetLanguagesSupported

This function gets a list of languages supported by a specified service of the plug-in. The service type is defined as LM_ServiceType. Languages are defined as the LM_LanguageCode, which is the ISO language code. See [“Language code” on page 25](#).

Syntax:

```
HRESULT GetLanguagesSupported ( SHORT inServiceType, SHORT *outCount, char**  
*outArray );
```

TABLE 6 *GetLanguagesSupported parameter table*

Name	Use	Description
inServiceType	[in]	The specified service type to get the language list for.
outCount	[out]	A count of the services.
outArray	[out]	An array of services for Linguistic Library to pick up.

ILiloSpellChecker interface

ILiloSpellChecker provides six functions for spelling services:

- IsWord() checks the spelling of a word.
- CorrectWord() returns an array of suggestions for a word.
- GetLanguageRef() returns an integer referencing a language, which is required by both IsWord() and CorrectWord().
- GetPropertyList(), GetProperty(), and SetProperty() deal with provider-specific properties (or capabilities/options).

The interface is defined as follows:

```
class ILiloSpellChecker : public IUnknown
{
public:
virtual /* [helpstring] */ HRESULT STDMETHODCALLTYPE GetLanguageRef (
/* [in] */ const char* inLanguage,
/* [retval][out] */ LONG *outLanguageRef) = 0;

virtual /* [helpstring] */ HRESULT STDMETHODCALLTYPE IsWord (
/* [in] */ BSTR inWord,
/* [in] */ LONG inLanguageRef,
/* [retval][out] */ VARIANT_BOOL *outResult) = 0;

virtual /* [helpstring] */ HRESULT STDMETHODCALLTYPE CorrectWord (
/* [in] */ BSTR inWord,
/* [in] */ LONG inLanguageRef,
/* [out] */ SHORT *outCount,
/* [out] */ BSTR **outArray) = 0;

virtual /* [helpstring] */ HRESULT STDMETHODCALLTYPE GetPropertyList (
/* [out] */ SHORT *outCount,
/* [out] */ SHORT** outPropertyArray,
/* [out] */ LONG** outValueArray) = 0;

virtual /* [helpstring] */ HRESULT STDMETHODCALLTYPE GetProperty (
/* [in] */ SHORT inProperty,
/* [out] */ LONG *outValue) = 0;

virtual /* [helpstring] */ HRESULT STDMETHODCALLTYPE SetProperty (
/* [in] */ SHORT inProperty,
/* [in] */ LONG inValue) = 0;
};
```

GetLanguageRef

This function gets the language reference of a given language code. This is for Linguistic Library to get a quick reference for the other function calls in this interface. The plug-in does not have to do a language ISO look-up when it receives a request. For some providers, this function can be used to determine the language currently used and to open the corresponding lexicon.

Syntax:

```
HRESULT GetLanguageRef ( const char* inLanguage, LONG *outLanguageRef );
```

TABLE 7 *GetLanguageRef* parameter table

Name	Use	Description
inLanguage	[in]	Language ISO code.
outLanguageRef	[out]	A reference to the given language.

IsWord

This function is used by Linguistic Library to verify the spelling of a word. It takes a word and returns true if the word is spelled correctly; otherwise, false.

Syntax:

```
HRESULT IsWord ( BSTR inWord, LONG inLanguageRef, VARIANT_BOOL *outResult );
```

TABLE 8 *IsWord* parameter table

Name	Use	Description
inWord	[in]	The input word
inLanguageRef	[in]	A reference to the given language.
outResult	[out]	true or false, if the word is spelled correctly.

CorrectWord

This function is used by Linguistic Library to get a list of suggestions for a misspelled word. It takes a word and returns a list of suggestions and a count of suggested words.

Syntax:

```
HRESULT CorrectWord ( BSTR inWord, LONG inLanguageRef, short *outCount, BSTR **outArray );
```

TABLE 9 *CorrectWord* parameter table

Name	Use	Description
inWord	[in]	The input word
inLanguageRef	[in]	A reference to the given language.
outCount	[out]	A count of suggested words.
outArray	[out]	An array of suggestions.

GetPropertyList

This function is used by Linguistic Library to obtain a list of capabilities supported by a plug-in. In some plug-ins, capabilities may be called options or properties. Linguistic Library also obtains a list of their values through this function. A value of 0 means false; 1, true. In the future, the values might be expanded to accommodate other property types.

Syntax:

```
HRESULT GetPropertyList ( SHORT *outCount, SHORT **outPropertyArray, LONG **outValueArray );
```

TABLE 10 *GetPropertyList* parameter table

Name	Use	Description
outCount	[out]	The number of properties supported by the plug-in.
outPropertyArray	[out]	An array of property type, the same as the ones defined as LM_CapabilityType.
outValueArray	[out]	Current values of each property.

GetProperty

This function is used by Linguistic Library to obtain the value of a capability. A value of 0 means false; 1, true.

Syntax:

```
HRESULT GetProperty ( SHORT inProperty, LONG* outValue );
```

TABLE 11 *GetProperty* parameter table

Name	Use	Description
inProperty	[in]	The capability of the value requested.
outValue	[out]	The value of the capability.

SetProperty

This function is used by Linguistic Library to set the value of the given capability. A value of 0 means false; 1, true.

Syntax:

```
HRESULT SetProperty ( SHORT inProperty, LONG inValue );
```

TABLE 12 *SetProperty* parameter table

Name	Use	Description
inProperty	[in]	The capability whose value will be set.
inValue	[in]	The value to set for the capability.

ILiloHyphenation interface

ILiloHyphenation provides three functions for hyphenation services:

- CutWord() divides a word.
- GetHyphenLanguageRef() returns an integer referencing a language.
- GetHyphenationPoints() returns hyphenation points.

The interface is defined as follows:

```
class ILiloHyphenation : public IUnknown
{
public:

virtual /* [helpstring] */ HRESULT STDMETHODCALLTYPE
GetHyphenLanguageRef(
/* [in] */ CHAR *inLanguage,
/* [retval] [out] */ LONG *outLanguageRef) = 0;

virtual /* [helpstring] */ HRESULT STDMETHODCALLTYPE
GetHyphenationPoints(
/* [in] */ BSTR inWord,
/* [in] */ LONG inLanguageRef,
/* [out] */ SHORT *outCount,
/* [out] */ SHORT **outPointArray,
/* [out] */ SHORT **outPreferenceArray) = 0;

virtual /* [helpstring] */ HRESULT STDMETHODCALLTYPE CutWord(
/* [in] */ BSTR inWord,
/* [in] */ LONG inLanguageRef,
/* [in] */ SHORT *inPointArray,
/* [in] */ SHORT inTargetPoint,
/* [out] */ BSTR* outLeftPart,
/* [out] */ BSTR* outRightPart) = 0;

};
```

GetHyphenLanguageRef

This function gets the language reference of a given language code. This is for Linguistic Library to get a quick reference for the other function calls in this interface.

Syntax:

```
HRESULT GetHyphenLanguageRef ( CHAR* inLanguage, LONG *outLanguageRef );
```

TABLE 13 *GetHyphenLanguageRef* parameter table

Name	Use	Description
inLanguage	[in]	Language ISO code.
outLanguageRef	[out]	A reference to the given language.

GetHyphenationPoints

This function returns a list of hyphenation points and corresponding preference values for a given word.

Syntax:

```
HRESULT GetHyphenationPoints ( BSTR* inWord, LONG inLanguageRef, SHORT *outCount,
SHORT **outPointArray, SHORT **outPreferenceArray );
```

TABLE 14 *GetHyphenationPoints parameter table*

Name	Use	Description
inWord	[in]	The word to be hyphenated
inLanguageRef	[in]	The language reference for the given word.
outCount	[out]	The number of hyphenation points.
outPointArray	[out]	An array of hyphenation point, 1-based index.
outPreferenceArray	[out]	An array of hyphenation preference values.

CutWord

This function is used by Linguistic Library to obtain the left and right parts of a hyphenated word. It is necessary to have this function as in some languages, the spelling of a word is changed after it is hyphenated.

Syntax:

```
HRESULT CutWord ( BSTR inWord, LONG inLanguageRef, SHORT *inPointArray, SHORT
inTargetPoint, BSTR* outLeftPart, BSTR* outRightPart );
```

TABLE 15 *CutWord parameter table*

Name	Use	Description
inWord	[in]	The word to be broken at a given hyphenation point.
inLanguageRef	[in]	A reference to the given language.
inPointArray	[in]	The array of hyphenation points, from the function GetHyphenationPoints.
inTargetPoint	[in]	The index point where the word needs be separated.
outLeftPart	[out]	The left part of the hyphenated word.
outRightPart	[out]	The right part of the hyphenated word.