Developing Desktop Applications

Livescribe™ Desktop SDK
Version 0.7.0

Copyright and Trademarks

DevDesktopApps-0.7.0-DesktopSDK-0.7.0-REV-F

Contents

Preface

About this Document

This document, *Developing Applications with the Livescribe Desktop SDK*, describes how to create custom Livescribe desktop applications in C#.

To familiarize yourself with the Livescribe Developer Documentation set, see *Using Developer Documentation* in the `Docs` directory of the Desktop SDK. If you have questions about Livescribe Platform terminology, see *Introduction to the Livescribe Platform* and *Developer Glossary* in the `Docs` directory of the Desktop SDK.

Introduction

We are proud to present the first release of the Livescribe Desktop SDK to the Livescribe Developer Community. As a developer of Livescribe smartpen applications, or penlets, you can now create a custom desktop application to provide off-pen support for your penlets.

A desktop application runs on a customer's laptop or desktop computer and leverages its computing power and display capabilities. The application retrieves strokes and audio from the smartpen and then processes, manipulates, and transforms the data, creates different views of data, and presents a user interface where customers can review, search, edit, annotate, and share.

A custom desktop application supplements the standard Livescribe Desktop application published by Livescribe, Inc. If your customers need only to back up their smartpen data, view and search notebooks, and listen to their audio recordings, Livescribe Desktop will suit them admirably. If you wish to perform special processing on the data generated by your penlet, present it in a novel fashion, or provide advanced editing capabilities, you should consider creating a custom desktop application.

The Livescribe Desktop SDK also provides a programmatic interface for creating Livescribe paper products. Although you can still create custom paper products for your penlet using the visual layouts of the Paper Editor, you may prefer the granular control offered by the paper product API.

Supported Development Platforms

The Livescribe Desktop SDK supports the following development platforms:

- Windows XP SP2/SP3
- Windows Vista
- Windows 7

Features of Livescribe Desktop SDK 0.7.0

The Livescribe Desktop SDK 0.7.0 allows you to create desktop applications that do the following:

- Access data generated by penlets that are installed on one or more Livescribe smartpens attached to your desktop computer.
- Access strokes stored in AFDs that are installed on one or more Livescribe smartpens attached to your desktop computer.
- Get smartpen status information, such as battery and memory status.
- Get and set smartpen properties.
- Manipulate data in ZIP files on the desktop computer.
- Create a paper product programmatically.

Contents of the SDK

The Livescribe Desktop SDK includes the Livescribe PenComm API, PenData API, AFP API, documentation, and sample code. All code is in C#.

Directory Structure of the SDK

The top level of the Livescribe Desktop SDK contains the following directory structure:

- **Bin**

  Dynamic Link Libraries (DLLs) required for developing C# applications with the Livescribe Desktop SDK.
- **Docs**
  - *APIDocs*: HTML-based reference for the PenComm API and Pen Data API. You can access both API reference sets by double-clicking `index.html`.
  - *Developing Desktop Applications*: PDF providing an overview on how to develop desktop applications in C# that access data from penlets installed on a Livescribe smartpen.
  - License documents, development and marketing guidelines, Livescribe Platform Introduction, Glossary, and Release Notes.
- **Samples**
  - **DesktopApps/common**

    Dynamic link libraries (DLLs) required for compiling sample projects into .NET assemblies.
  - **DesktopApps/cs**

    Source and project files for creating a sample desktop application in C# with Microsoft Visual Studio 2008.
  - **Penlets**

    Source and project files for creating a sample penlet in Java with Eclipse.

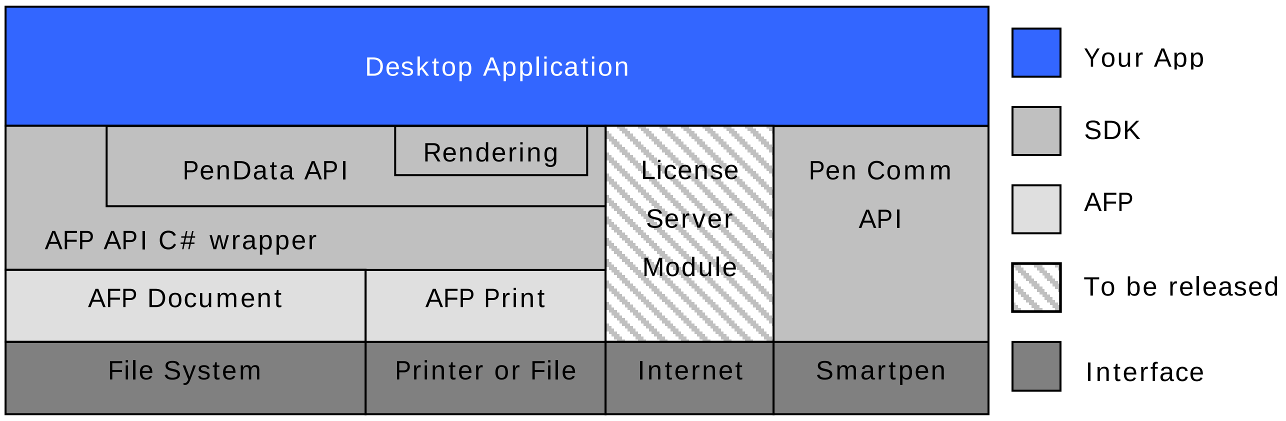
Understanding Livescribe Desktop SDK

The Livescribe Desktop SDK provides a C# Programming Interface (API) for developing applications that interact with the Livescribe smartpen. Basic interactions include smartpen status information, penlet data transfer, and stroke data transfer.

Customers can update smartpen firmware with Livescribe Desktop only. Firmware updating requires central management by the Livescribe server to ensure that the smartpen remains in a consistent state. Consequently, custom desktop applications cannot access the smartpen's firmware updating mechanism.

Overview of Architecture

The following diagram describes the architecture of the Livescribe Desktop SDK.



Desktop SDK Overview

API in the Livescribe Desktop SDK

PenComm API

The PenComm API allows your desktop application to connect to one or more smartpens to perform the following tasks:

- Get smartpen information.
- Detect data changes.
- Transfer non-stroke penlet data from a smartpen and write the data to a ZIP file on the desktop computer's file system.

- Perform smartpen commands and penlet commands.

PenData API

The PenData API allows your desktop application to access the penlet data or AFDs and perform the following tasks:

- Retrieve non-stroke penlet data.
- Retrieve stroke data from AFDs.
- Render AFDs on a desktop computer monitor.
- Create AFDs programmatically.

AFP API Wrapper

The AFP API Wrapper is a C# wrapper for the Print and Document modules of the AFP (Anoto Functionality Platform) library. It is the low-level API upon which all AFD access is based. For most coding, however, you will use the higher level PenData API instead, since it provides simpler access with less coding. Certain tasks, however, require that you use both APIs. If your application has sophisticated or highly sensitive performance needs, you will make extensive use of the AFP API Wrapper.

Communicating with a Smartpen

PenComm Service

To develop and run desktop applications, the PenComm Service must be installed on your desktop computer. The PenComm Service is installed automatically with the Livescribe Desktop software. Update your Livescribe Desktop to the latest version to ensure that you obtain the most up-to-date PenComm Service.

If your computer does not have Livescribe Desktop installed, you can install the PenComm Service using the Microsoft Windows installer that is included in the Livescribe Desktop SDK. The path in the Desktop SDK is:
`Bin/cs/PenComm/PenCommServiceInstaller.msi`

Overview of Desktop Application and Smartpens

A desktop application can communicate with smartpens that a user attaches to the desktop computer. To do so, your application should implement the following steps.

Note: The terms *attaching* a smartpen and *docking* a smartpen are synonymous.

1. Create a `Smartpens` object.

   Creating a Smartpens object automatically initializes the PenComm Client Library and allocates resources for communicating with attached smartpens.

2. To specify a custom log file, pass `true` as the first parameter to the Smartpens construct, and the custom log file path as the second parameter. For example:

   ```
   Smartpens smartpens = new Smartpens(true, "CSSampleDesktApp.log");
   ```

   - The default log file name is `PenCommSdk.log` and the default location is the directory that contains the PenComm Service executable.

3. To be notified asynchronously that a smartpen has been docked, register a smartpen attach event handler. For details, see Docking a Smartpen.

4. To be notified asynchronously when a smartpen is undocked, register a smartpen detach event handler. For details, see Undocking a Smartpen.

5. You can receive data *synchronously* from a smartpen. Call the `DataGet` method on your `Smartpen` object. A common place to make the call is from within the smartpen attach event handler. The `DataGet` method specifies the package (penlet or AFD) from which you wish data.
6. You can receive *asynchronous* notification from the PenComm Service that one or more packages have new data. These are **packages of interest** for your desktop application. They can be penlets or AFDs.

   Implementing asynchronous handling of new data involves:

   a. Registering your desktop application.

   b. Registering one or more packages of interest for your desktop application.

   c. Implementing a data callback that handles new data from these packages of interest.

   d. Registering the data callback.

   For more detail, see Registering Packages of Interest and Data Callback.
7. Trigger the re-enumeration of smartpens by calling the `Find` method on your `Smartpens` object.

Getting Incremental Data

When requesting data from a smartpen, your desktop application needs to know when the data was created. We recommend that you request existing data the first time the smartpen is docked. Thereafter, your application may choose to receive only data created since the last time the smartpen was docked. The original data set plus all increments constitute the complete data set for the smartpen.

You have two choices for obtaining increment of data:

- Synchronously.
- Asynchronously.

Start From Time

If you get your data increment synchronously, you must specify the *start from time* of the data. Only data created on the smartpen after the start time will be delivered to your application. Pass that start time to the `SmartpenChangeList.Update` and the

`Smartpen.DataGet` methods. For details, see The DataGet Method and Smartpen Change List.

Last Docking Time

If you receive your data asynchronously, you automatically get the data increment since the *last dock time*—that is, the time that the smartpen was last docked to the desktop computer. Every time the smartpen is docked, the PenComm Service:

- Passes the smartpen attach event handler a `Smartpen` object with a `ChangeList` property. The PenComm Service calls the registered smartpen attach event handler and passes a `Smartpen` object. The `ChangeList` property of the `Smartpen` object is a `SmartpenChangeList` object. The change list will be empty until you call the `ChangeList.Update` method on the `Smartpen` object, passing a start time parameter.
- Invokes your registered data callback with data created since the last docking time.

When a smartpen is docked, the PenComm Service records the current smartpen RTC as the *dock time*. The increment of new data is the data created on the smartpen between the *last dock time* and the *dock time*.

Merging the Retrieved Data

Whether retrieved synchronously or asynchronously, the increments must be merged programmatically by your desktop application. To simplify this task, save data increments to files and directories whose names indicate the source of the data—for example, filenames that indicate the smartpen ID, package name, retrieval time, etc. See Merging AFDs.

Docking a Smartpen

When the user attaches a smartpen to the desktop computer, the PenComm Service performs the following actions, and your desktop application should respond as indicated. If the service has connected with the smartpen before, it is considered an *existing* smartpen. Otherwise, it is a *new* smartpen.

1. The PenComm Service detects that a smartpen has been attached and determines whether it is a new or existing smartpen. The service sets the *dock time* value to the current RTC time from the smartpen.
   - For a new smartpen, the PenComm Service considers the *last dock time* to be zero (0), so that your application receives all current data from the smartpen.
   - For an existing smartpen, the *last dock time* has the value of *dock time* from the last occasion the smartpen was docked.
2. The Livescribe Desktop Launcher is listening for smartpen attach and smartpen detach events. If a desktop application is configured to auto-start and is not running, then the Livescribe Desktop Launcher starts it.
3. The PenComm Service sends an event to all registered desktop applications by calling their registered smartpen attach event handler. To register the handler:
   a. Implement a smartpen attach event handler whose signature matches the following delegate type, declared in the `Smartpens` class:

      ```
      public delegate void SmartpenChangeCallback(Smartpen pen)
      ```

   b. Create an instance of that delegate type.

   c. Add the delegate instance to the `SmartpenAttachNormalEvent`, defined in the `Smartpens` class.

   Registration code for a smartpen attach event handler will look something like this (assuming your handler is named `PenAttachEvent`):

   ```
   smartpens.SmartpenAttachNormalEvent += new
   Smartpens.SmartpenChangeCallback(PenAttachEvent);
   ```

4. The smartpen attach event handler should internally track the event information. In the `Smartpen` parameter of the event handler, the PenComm Service passes an object representing the smartpen that was just attached.

   The `Smartpen` object has three important properties that your desktop application may wish to access from the handler:

   - **ChangeList:** a `SmartpenChangeList` object that wraps a collection of `ChangeItem` objects. Each `ChangeItem` object contains information on changes in a package that occurred since the last dock time. You can use this information to request the actual data.

  Note that the `SmartpenChangeList` object will be empty until you call `ChangeList.Update` on the `Smartpen` object, passing a start time parameter.

- **Hardware:** a `SmartpenHardware` object containing properties such as battery level, memory status, Pen ID (a unique integer), Pen Serial (the Pen ID as a string), smartpen firmware version, etc.
- **Packages:** a `SmartpenPackages` object that wraps `PackageItem` objects. Each `PackageItem` object represents information about a penlet or an AFD installed on the smartpen.

5. **Optionally:** the smartpen attach event handler may also register the data callback with the PenComm Service. See Registering Packages of Interest and Data Callback.
6. The PenComm Service checks registered packages of interest (if any) for data that is new since the *last dock time*. The service ignores any package that has not been registered as a package of interest.
7. The PenComm Service attempts to transfer a package's new data by invoking the desktop application's data callback at least once for each package that has new data.
   a. The transfer operation fails if your desktop application is not currently running. In that case, the service sends the data to your application later—after your application starts and registers the data callback.
8. The PenComm Service sets the *last dock time* value so it is ready for the next occasion when the smartpen is docked: the current *dock time* value is assigned to *last dock time*.

Undocking a Smartpen

When a smartpen is detached (or, *undocked*), the PenComm Service takes the following actions, and your application should respond as indicated.

1. The PenComm Service detects that a smartpen has been detached.
2. The PenComm Service sends an event to all registered desktop applications by calling their registered smartpen detach event handler. To register the handler:

a. Implement a smartpen detach event handler whose signature matches the following delegate type, declared in the `Smartpens` class:

   `public delegate void SmartpenChangeCallback(Smartpen pen)`

b. Create an instance of that delegate type.

c. Add the delegate instance to the `SmartpenDetachNormalEvent`, defined in the `Smartpens` class.

Registration code for a smartpen attach event handler will look something like this (assuming your handler is named `PenDetachEvent`):

```
smartpens.SmartpenDetachNormalEvent += new
Smartpens.SmartpenChangeCallback(PenDetachEvent);
```

3. Your desktop application should respond appropriately. For instance, you might remove this smartpen from the list of smartpens displayed in the UI of your application.

Sample Application: Getting Smartpen Status

Your desktop application can access the status information of smartpens attached to the computer. Use the `Hardware` property of the Smartpen object passed to you by the smartpen attach event handler.

The following sample demonstrates how to access the smartpen's battery and memory status, as well as the user time. It is a fully-functioning example of a very simple desktop application, showing how to initialize the PenComm Client Library, create a custom log file, handle smartpen attach and detach events, and display the user time in a human-readable format.

```
using System;
using System.Threading;
using Livescribe.DesktopSDK.PenComm;

namespace GetPenInformation {
    class GetPenInformation {
        private const string PENCOMM_LOG = "GetPenInformation.log";
        static bool foundPen = false;

        static void Main(string[] args) {
            Smartpens smartpens = new Smartpens(true, PENCOMM_LOG);

            // The Smartpen object keeps track of attach event handlers in
            // SmartpenAttachNormalEvent
            // Create a new callback object for our PenAttachEvent method and
            // register it by adding it to the list of attach event handlers.
            smartpens.SmartpenAttachNormalEvent +=
               new Smartpens.SmartpenChangeCallback(PenAttachEvent);

            // Do the same for our detach event handler.
            smartpens.SmartpenDetachNormalEvent +=
               new Smartpens.SmartpenChangeCallback(PenDetachEvent);

            Console.Write("Waiting for pen.");

            // Search for any already docked pens
            smartpens.Find();

            // Keep processing until user presses a key to terminate
            while (!Console.KeyAvailable) {
                if (!GetPenInformation.foundPen) {
                    Console.Write(".");
                }
                Thread.Sleep(2000);
            }
        }

        static private string FormatMem(ulong bytes) {
            return (bytes / (1024 * 1024)) + " MB";
        }

        static private string FormatUserTime(ulong userTime) {
            // Create DateTime object that represents 1/1/1970
            DateTime unixEpochTime = new DateTime(1970, 1, 1);

            // Adding userTime (the number of milliseconds since 1/1/1970)
            // gives us the current time.
            DateTime currentPenTime = unixEpochTime.AddMilliseconds(userTime);

            return currentPenTime.ToString();
        }

        static private void PenAttachEvent(Smartpen pen) {
            GetPenInformation.foundPen = true;

            // The pen.Hardware property is not valid until we update it.
            pen.Hardware.Update();

            // Write out some interesting information about the connected pen
            Console.WriteLine("\n\n== Pen " + pen.PenSerial + " connected ==");
            Console.WriteLine("Model = " + pen.PenTypeText);
            Console.WriteLine("Battery Level = " + pen.Hardware.Battery.Voltage + " V");
            Console.WriteLine("RTC Time = " + pen.RtcTime());
            Console.WriteLine("User Time = " + FormatUserTime(pen.UserTimeGet()));

            Console.WriteLine("\n== PenMemory == ");
            Console.WriteLine("Apps   = " + FormatMem(pen.Hardware.MemoryApplication));
            Console.WriteLine("Data   = " + FormatMem(pen.Hardware.MemoryFiles));
            Console.WriteLine("System = " + FormatMem(pen.Hardware.MemorySystem));
```

```
            Console.WriteLine("Free    = " + FormatMem(pen.Hardware.MemoryFree));
            Console.WriteLine();
        }

        static private void PenDetachEvent(Smartpen pen) {
            Console.WriteLine("\n== Pen " + pen.PenSerial + " disconnected ==");
        }
    }
}
```

Registering Desktop Applications

You can register your desktop application with the PenComm Service. There are several reasons for doing so:

- To instruct the PenComm Service that your desktop application should be automatically started when a smartpen is docked and/or packages of interest have new data. (Of course, this option applies only if your desktop application is not already running.)
- As the first step in receiving data via a data callback.
- So any other Livescribe desktop applications can get a list of all registered applications on the computer.

To register the desktop application with the PenComm Service, do the following:

1. Create a `RegAppInfo` object and set the following members:
    - **autoStart**—Constant indicating the conditions under which your desktop application should be started automatically by the PenComm Service. Possible constants are:

Constant	**Description**
NotSet	Auto Start not configured
StartAlways	Always start application when a smartpen is docked
StartNewDataOnly	Start application only when a package of interest has new data

StartNewDataOnlyOrNewPen	Start application when a package of interest has new data or when a new smartpen (i.e., **not** an existing smartpen) is docked

- **description**—String describing your desktop application.
- **version**—String indicating the version number of your desktop application.
- **path**—String specifying the full path to your desktop application.

2. Call the `Smartpens.DesktopApplications.RegisterApp` method, passing the `RegAppInfo` object.

   The method returns the application handle for your desktop application. You will use the application handle when registering packages of interest and a data callback.

Registering Packages of Interest and Data Callback

To receive new data from a package on a docked smartpen, your desktop application must register its interest in these packages. They become *packages of interest* for your application. In addition, your desktop application must register a callback that the PenComm can whenever it detects new package data. This occurs most commonly when the user attaches a smartpen to the desktop computer. Scenarios for new penlet data being detected while the smartpen is attached to the computer are less common at present.

Setting up a data callback involves four steps:

1. Register your desktop application. See Registering Desktop Applications, above.
2. Register your package(s) of interest. See below.
3. Implement a data callback. See Data Callback.
4. Register the data callback. See below.

Where to Register

There are two main places in your code where you perform the registration of your application, your package(s) or interest, and your data callback:

- In the smartpen attach event handler.
- In your PenComm library initialization code—after you create the `Smartpens` object (which initializes the library) and before you add the smartpen attach event handler to the `Smartpens.SmartpenAttachNormalEvent`, The following code snippet demonstrates the location:

```
smartpens = new Smartpens(true, PENCOMM_LOG);

//YOU CAN INSERT APP, PACKAGE, DATA CALLBACK REGISTRATION HERE

smartpens.SmartpenAttachNormalEvent += new
   Smartpens.SmartpenChangeCallback(PenAttachEvent);

smartpens.Find();
```

Registering Packages of Interest

To register a package as a *package of interest* for your desktop application, do the following:

1. Create a `RegPackageInfo` object and set the following members:
    - **appHandle**—The application handle for the desktop application that is registering the package. You obtained this handle when you called the `Smartpens.DesktopApplications.RegisterApp` method to register your desktop application.
    - **penId**—The unique 64-bit unsigned integer that identifies a smartpen. It is a `Livescribe.DesktopSDK.PenComm.PenId` object. You can indicate that your desktop application is interested in all attached smartpens by setting this member to 0.
    - **uniqueName**—String uniquely identifying the package. For a penlet, it is the fully-qualified class name. For an AFD, it is the GUID.
2. Call the `Smartpens.DesktopApplications.RegisterPackage` method, passing the `RegPackageInfo` object.

The method returns the package handle for your package. You will use the package handle in your implementation of the data callback.

Registering the Data Callback

To register your data callback, do the following:

1. Call the `Smartpen.DesktopApplictions.RegisterDataCallback` method, passing in the name of your data callback implementation.

For information on how to implement your data callback, see Data Callback. For a working sample of a data callback, see Penlet Data Retrieval Example.

Listing Applications and Packages

You can get various lists of Livescribe desktop applications and packages.

Listing Registered Applications

The PenComm Service can handle multiple Livescribe desktop applications running simultaneously. To get a list of the other desktop applications that are registered with the PenComm Service, do the following:

1. Call the `Smartpens.DesktopApplications.GetRegisteredAppList` method.

It will return a list of `RegAppInfoItem` objects. Each object has properties that return the application handle and the `RegAppInfo` object that was used to register the application.

Listing Registered Packages of Interest

To get a list of packages that are packages of interest to one or more desktop applications on your computer, do the following:

1. Call the `Smartpens.DesktopApplications.GetRegisteredPackageList` method.

It will return a list of RegPackageInfoItem objects. Each objects has properties that return the package handle and the `RegPackageInfo` object that was used to register the package.

Listing All Packages on a Smartpen

To get a list of all packages that are installed on a smartpen, do the following:

1. Read the `Packages` property of the `Smartpen` object passed by the PenComm Service to your smartpen attach event handler.

This property returns a collection of `PackageItem` objects, each of which contains information about a package, such as:

Property	**Description**	**Examples**
PackageName	user-friendly name of the package	Paper Replay Flip Notebook 1
FullPath	location of the package on the smartpen	**penlets/LS_Paper_Replay.jar** **penlets/LS_Flip_Notepad_01_pen.af**
Version	version of package	**1.0** **2.1** **2.3**
ClassName	ID of the package	
	For a penlet— the fully-qualified class name.	**com.livescribe.paperreplay.PaperReplay**
	For an AFD— the GUID	**0x3ce96ea355e05d59**

To retrieve a list of packages for a smartpen, you can code the following in your smartpen attach event handler:

1. Call `Packages.Update` method on the `Smartpen` object passed in the event handler.
2. Iterate over the `PackageItem` objects and do something appropriate. For example, you might wish to add them to a List, for display on the UI of your application.

The following sample code illustrates how you might code a smartpen attach event handler to retrieve the list of packages, and then call custom methods to output a list of document info (for an AFD) or the file and directory names of penlet data (for a penlet).

The relevant portion of the smartpen attach event handler looks like this:

```
private static void PenAttachEvent(Smartpen pen) {
   foundPen = true;
   try {
       // Refresh packages list
       pen.Packages.Update();

       // Show package data
       foreach (PackageItem packageItem in pen.Packages.Items) {
           System.Console.WriteLine("\n==============================");
           System.Console.WriteLine("\nPackage: " + packageItem.PackageName +
              " (" + packageItem.FullPath + ")");

           // Retrieve the data of the package
           pen.DataGet(packageItem.PackageName, 0, DATA_FILE);

           // Anything that comes from the pen can be opened as a container
           try {
               Container container = new Container(DATA_FILE);

               // If container is a document, we can access the document member
               if (container.IsDocument()) {
                  PrintAFDInfo(container.Document);
               }

               // If container is a penlet, we can access the penlet data member
               if (container.IsPenletData()) {
                  PrintPenletDataFilesDirs(container.PenletData);
               }
               container.Close();
           }//end inner try
       }//end foreach
. . .SKIPPING EXCEPTION CATCHING STATEMENTS HERE, FOR BREVITY . . .
  }//end outer try
}//end method
```

For the moment, you can ignore the `pen.DataGet` method and the Container class. Both will be discussed in subsequent sections of this manual. Also, in the interest of space, the custom methods `PrintAFDInfo` and `PrintPenletDataFilesDirs` are not reproduced here. If you wish to examine the entire ListPackages sample application, see Accessing an AFD.

Getting the Smartpen Data File on Your Desktop

The previous section has discussed how you can use the PenComm API to connect to docked smartpens, get smartpen status, list packages on a smartpen, and register your desktop application with the PenComm Service.

With those preliminaries out of the way, we will now address one of the most fundamental tasks of a desktop application: getting data from a smartpen. This section explains how to obtain data from a docked smartpen and write it to a compressed file (a ZIP file) on your computer. The starting point for all smartpen data manipulations is a ZIP file containing the smartpen data you retrieved. The smartpen data may be:

- Penlet Data: *Non-stroke, application-generated data* that was stored programmatically by a penlet to penlet storage on the smartpen. This data can include many files of different kinds, such as *audio files* if a penlet records sound, as Paper Replay does. It can include *text* generated by the penlet by invoking the HWR (Handwriting Recognition) engine on the user's strokes in real time. Or it can be *data calculated* by the penlet in response to user input. But to repeat: it does **not** include the strokes created by the user on Livescribe paper.
- AFD: a compressed file containing a paper product definition, dot pattern, strokes, and other data. The *paper product definition* contains page layouts, including the graphic images and static regions on each page layout. *Dot pattern* consists of a unique pattern page for each page of the paper product. *Strokes* are data describing the strokes made by smartpen users on each page of the paper product.

  Other data includes the same user-generated data you can access via Penlet Data. However, this store contains data from all penlets and smartpens that have used the paper product. If your paper product has Open Paper sections, much of that data may not be relevant to your desktop application. In the most common case, you should use `Smartpen.DataGet` and pass the name of your penlet.

When You Create a Smartpen Data File

There are two principal situations in which you create a smartpen data file on your desktop computer: (1) in response to a SmartpenAttachNormalEvent and (2) in a Data Callback.

Smartpen Attach Event

When the `SmartPenAttachNormalEvent` occurs, the PenComm Service invokes any methods that your desktop application registered as handlers. A `SmartpenAttachNormalEvent` is delivered for each smartpen attached; all registered methods for that event are called for each smartpen. Our samples have only one method registered to handle the smartpen attach event.

A smartpen attach event handler is passed the `Smartpen` object for which the event is being delivered. In the handler, you can:

- call the `DataGet` method on the Smartpen object to retrieve data from a package.
- register a data callback so that the PenComm Service can send your desktop application new data from packages of interest.

Sometimes, your smartpen attach event handler will do both.

The DataGet Method

When you wish to make a synchronous call to get data from a smartpen, invoke the `Smartpen.DataGet` method. It can be called on any package. The method is overloaded, but its most-used signature is:

```
public bool DataGet(string package, UInt64 time, string destFile)
```

Parameter	**Description**
Package	*For penlet*: fully-qualified classname
	For AFD: the GUID
Time	You are requesting data created after this time.

destFile	Full path to the ZIP file on your desktop. The transferred penlet data or AFD data will be stored in this ZIP file.

Start Time

When calling the `DataGet` method, you must also pass in a start time. To get just the new data generated by the package, pass a smartpen RTC time. To get all data generated by the package, pass a time of 0.

One approach to getting incremental package data is as follows: Call the method initially with a `time` of 0. For subsequent calls to `DataGet`, pass the smartpen RTC time of your previous call. Thus, on each call to `DataGet`, you should store the current smartpen time, so you can use it for your next call. You can access the current smartpen time by calling the `RtcTime` method on the `Smartpen` object.

To determine whether it is worth calling `DataGet` on a package, you will wish to know if it has new data. To get a list of penlets and AFDs with new data, use the `ChangeList` property of the `Smartpen` object passed to you by the smartpen attach event handler. Proceed as follows:

1. Call the `ChangeList.Update` method of the `Smartpen` object, passing in the start time.
2. Access the change list via the `ChangeList` property of the `Smartpen` object. Iterate over the `ChangeItem` objects in the change list. Each change item describes one package that has data changes.
3. For each package with data changes, call the `DataGet` method, passing in the same start time you passed to `SmartpenChangeList.Update`. (For more on change lists, see the section titled *Smartpen Change List*, below.)

Exception for Not Installed or Not Run Yet

If you call `DataGet` and pass a package that is not installed on the smartpen, you will get an exception. Your desktop application can inform the user that the AFD or penlet is not installed on the smartpen, or handle the absence in some other way.

If you call `DataGet` and pass a penlet that is installed but has **not been run** on the smartpen, you will also get an exception. Your desktop application can inform the user that the penlet must be run at least once on the smartpen before penlet data from it can be transferred.

Smartpen Change List

The smartpen change list returns a digest of the changes made to penlet data or AFDs since a given time. Change list items do not contain the actual penlet data changed or the strokes added. Instead, a change list item describes penlets or AFDs which have new data.

Generally, you access a change list from within the smartpen attach event handler, using the `Smartpen` object provided to you. The `ChangeList` property of the `Smartpen` object returns a collection of `ChangeItem` objects. Each `ChangeItem` object relates to a particular package that has changes.

After a smartpen is docked, the `ChangeList` property of the `Smartpen` object is initially empty. To retrieve the current change list, call the `ChangeList.Update` method on the `Smartpen` object, passing in the start time, which is a smartpen RTC time. You will get a list of penlets and AFDs that have data created or modified after the time specified.

If the package is a **penlet**, the change item indicates the class name of the penlet. You can use the class name to request the associated penlet data. The fully-qualified class name of the penlet is returned by the `Classname` property of the `ChangeItem` obejct. The `Guid` property (and the `Pages` property) will be null.

If the package is an **AFD**, the change item indicates the GUID of the AFD. You can use the GUID to request the data from the AFD. Such data includes strokes, if there are any. A string representing the GUID of the AFD is returned by the `Guid` property of the `ChangeItem` object. (The `Pages` property will also have a non-null value.) The `Classname` property will be null.

The `Classname` property and the `Guid` property are valid alternately: If one is set, the other is not. They are *never* both set in the same change list item.

To process the change list, iterate over the collection of `ChangeItem` objects. Test the `Classname` and `Guid` properties of each change list item to see if it refers to penlet data or an AFD.

You may be interested in penlet data from your penlet(s) only: For each change list item that denotes penlet data, check for one of your penlets by getting the `Classname` property. For each of your penlets, call `DataGet` on the `Smartpen` object and pass the `Classname` value. Also pass an appropriately-named ZIP file to hold the data streamed from the smartpen. An appropriate naming scheme might involve concatenating values from the `penSerial`, `ClassName`, and `EndTime` properties of the change list item.

You may be interested in retrieving strokes: For each change list item that denotes an AFD, check for one of your AFDs by obtaining the `Guid` for the AFD (or use the AFD's `Title` property). For each of your AFDs, call `DataGet` on the `Smartpen` object and pass the `Guid` value. Also pass an appropriately-named ZIP file to hold the AFD transferred from the smartpen. An appropriate naming scheme might involve concatenating values from the `penSerial`, `Guid`, and `EndTime` properties of the change list item.

A change item for an AFD also includes a `Pages` property, which returns a collection of `PageChangeItem` objects, each of which represents a page that has changes (including strokes) in the AFD. After you transfer the AFD data to your desktop computer, you can access the stroke information from individual pages, using the collection of `PageChangeItem` objects. See Using the Smartpen Change List in the Retrieving Strokes section.

Start Time

To populate the change list, you must call the `ChangeList.Update` method on the `Smartpen` object passed to you by the smartpen attach event callback. The change list will be updated, and the `StartTime` property in the `SmartpenChangeList` object will reflect the start time you passed to the `Update` method. You can then pass this `StartTime` to the `DataGet` method to retrieve the actual data.

The start time might be a value particular to your desktop application and the data it is retrieving. For instance, you may wish to get data starting from a particular milestone date, no matter how many times the smartpen has been docked in the interim. You must determine and store this time yourself.

If you wish to perform incremental data retrieval, then pass a value to the `time` parameter of the `SmartpenChangeList.Update` method that represents the smartpen RTC time of the *previous* occasion that you called `Update`. Your code is responsible for storing the smartpen RTC time of that previous call, and then loading and passing it to the current call.

Data Callback

When a package of interest has new data, the PenComm Services calls the registered data callback of your desktop application. For details on registering a data callback, see Registering Packages of Interest and Data Callback.

The data callback you implement must match the signature of `PenAPI.DataCallback`:

```
public delegate PenAPI.DataOpTypes DataCallback ( PenDataCallbackInfo dataInfo,
            IntPtr userParam);
```

The PenComm Service returns essential information in the `dataInfo` parameter, which is a `PenDataCallbackInfo` structure. The structure includes the following members:

Member	**Description**
penId	The ID of the smartpen. It is a 64-bit unsigned number.
packageHandle	The package handle of the package that the transferred data is from. You can pass the **`penletHandle`** to the **`ApplicationManagement.GetRegisteredPackage`** method, which returns the **`RegPackageInfo`** object. That object contains the class name or GUID of the package.
startRtc	The start time of the data being transferred by this callback. This data was created on the smartpen **after** this time. It is generally, the *last dock time* of the smartpen.
buffer	The actual chunk of data being transferred in this invocation of the data callback.
bufSize	Size in bytes of the current chunk of transferred data.
packetStatus	Constant representing the status of the chunk of data transferred in the **`buf`** member. Defined in the **`PenAPI.DataPacketStatus`** enumeration, the possible constants are:
	MorePackets — There are more chunks. They will be sent in subsequent invocations of the data callback.
	LastPacket — The current chunk is the final one of the current file being transferred from the smartpen.
	LastPacketAndLastFile — The current chunk is the final chunk of the final file being transferred from the smartpen.

Generally, your data callback should proceed as follows:

1. Check the `packageHandle` member of the structure to determine which package of interest this invocation of the data callback concerns. You can perform any special processing, if this package requires it.

2. Read the **buffer** member of the structure into a local buffer and write it to a ZIP file on the computer's file system. Give the ZIP file an appropriate name, perhaps concatenating the `penId`, `penletHandle`, and `startRtc` values of the structure.

The following is a sample data callback named `NewData`. In particular, note the various uses of the `dataInfo` parameter.

```
private static PenAPI.DataOpTypes NewData(PenDataCallbackInfo dataInfo,
           IntPtr userParam) {
       try {
          // Construct file name that will be unique to a single stroke. We will save our
          // data in that file. We need to save the data in a file in case
          // the stroke is transferred to the desktop app in multiple invocations of the
          // callback.  This may happen, since a stroke can be composed of many
          // co-ordinates.
          String filePath = Path.GetTempPath() + "Pen" + dataInfo.penId +
               "_Penlet" + dataInfo.penletHandle + "_Time" + dataInfo.startRtc + ".zip";

          // Create or append to the file, depending on whether we're
          // starting with the first co-ordinate
          BinaryWriter bw = null;
             if (dataInfo.pos == 0) {
                bw = new BinaryWriter(new FileStream(filePath, FileMode.Create));
             }
             else {
                bw = new BinaryWriter(new FileStream(filePath, FileMode.Append));
                }

                byte[] buf = new byte[dataInfo.bufSize];
                Marshal.Copy(dataInfo.buffer, buf, 0, dataInfo.bufSize);
                bw.Write(buf);
                bw.Close();

                if (dataInfo.packetStatus != PenAPI.DataPacketStatus.MorePackets) {
                   PenletData penletData = new Container(filePath).PenletData;
                   DisplayData(penletData);
                }
             }
       catch (Exception e) {
          HandleError(e);
       }
       return PenAPI.DataOpTypes.Continue;
    }
```

Smartpen Attach Event Handler vs. Data Callback

You may be wondering when to implement a smartpen attach event handler and when to implement a data callback. As a rule of thumb, a smartpen attach event is best when you want to process data from multiple smartpen packages. On the other hand, a data callback is simpler and more efficient when processing data—particularly, a lot of data—from a single smartpen package.

Tip: Remember that your smartpen attach event handler should return relatively quickly. Consequently, calls to `DataGet` that involve large data transfers should be done in a separate thread—not in the attach event handler.

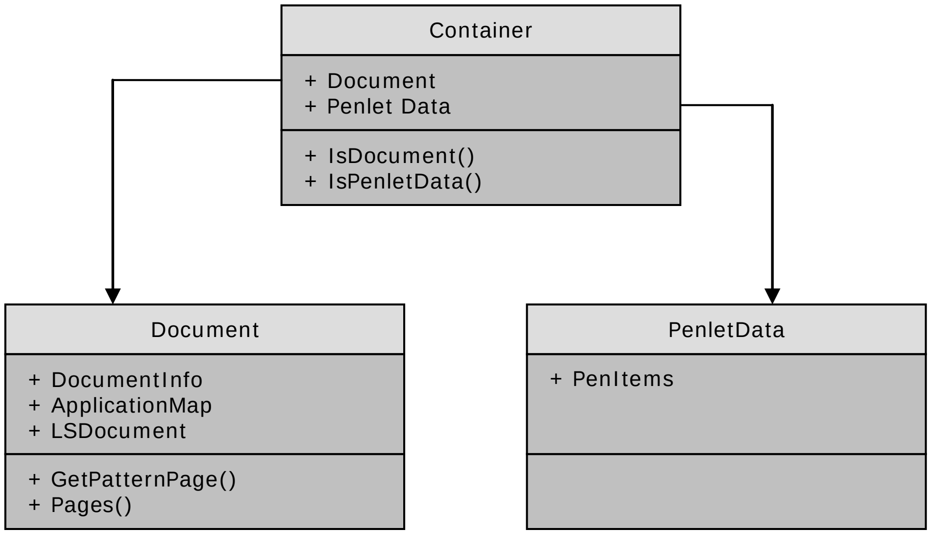
The Container

Once you have a ZIP file containing Penlet Data or an AFD, your desktop application is ready to use the PenData API to access that file and do interesting things with it.

The Container class represents the ZIP file on your desktop computer. In fact, the most common constructor of Container takes the file path to your ZIP file. Your first question is whether this is penlet data or an AFD. The `IsPenletData` and `IsDocument` methods provide the answer.

The following high-level diagram shows the relationship between the `Container`, `Document` and `PenletData` objects. Many important methods and properties are omitted.

Note: This class diagram, and the others in this manual, depicts containment hierarchies of objects. It is *not* a generalization hierarchy of superclasses and subclasses. These object hierarchies are created automatically for you by the Desktop SDK APIs and serve to model various aspects of smartpen communication and smartpen data access.



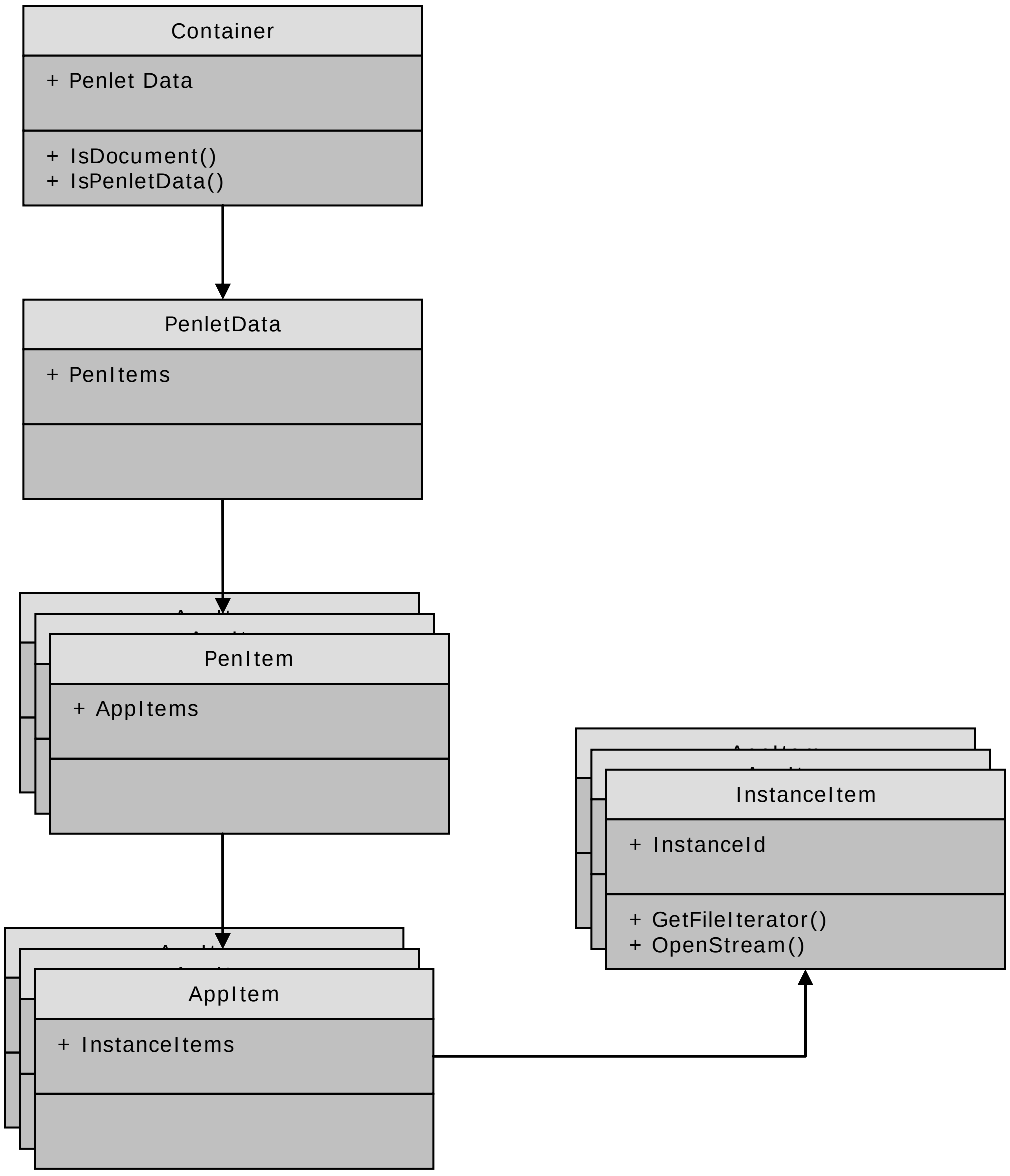
If your `Container` has penlet data, you will use the `PenletData` object. The Document property will be null. The most important property of the `PenletData` object is the `PenItems` property, which will allow you to drill down into the Penlet Data object model.

If your `Container` has an AFD, you will use the `Document` object. The PenletData property will be null. The most important members of the `Document` object are:

Member	**Description**
Pages	Returns a collection of the pages in the AFD.
GetPatternPage	Returns the pattern page (page instance) of a page in the AFD. A **`PatternPage`** object gives you access to the strokes on a page.
DocumentInfo	Metadata on the AFD, such as the author and GUID.
ApplicationMap	A map of key-value pairs. The key is the Applications ID used in static regions to identify a penlet. The value is the penlet's fully-qualified classname. See Penlet vs. Instance. (Remember that static regions are defined in an AFD.)
LSDocument	Low-level object for manipulating documents. The LSDocument class is defined in the AFP API Wrapper. See LSDocument.

Accessing Penlet Data

To access penlet data, your desktop application uses the Penlet Data object model. The model starts with a ZIP file containing the penlet data. Through a hierarchy of contained objects, it offers access to the individual data files stored in the ZIP file. The hierarchy is automatically created by the PenData API when you create a Container object. Following is a very high-level diagram.



The desktop application code that accesses the model can look something like this snippet:

```
Container container = new Container(PENLET_DATA_ZIP_FILE);
PenletData penletData = container.PenletData;

if (!container.IsPenletData()){
    Console.Out.WriteLine("Not a penlet data file");
    return false;
}
foreach (PenItem penItem in penletData.PenItems){
    foreach (AppItem appItem in penItem.AppItems){
       foreach (InstanceItem instance in appItem.InstanceItems){
          //Use Penlet Data here in remarkable ways!
       }
    }
}
```

The `Container` object has a `PenletData` property, which returns the `PenletData` object for this ZIP file.

The `PenletData` class encapsulates the penlet data generated by one or more penlets on one or more smartpens. It is a wrapper for the `PenItems` collection and implements the `IEnumerable` interface.

The samples in this section assume that the penlet data is from one penlet on one smartpen. If you created your ZIP file by calling pen.DataGet and passing in your fully-qualified classname, this easy-to-handle case applies to your code.

On the smartpen, penlet data exists in penlet storage. The `smartpen.DataGet` call queries your penlet for the penlet data, and your penlet must serve it up by implementing the Remote interface. The data callback gets penlet data from penlet storage by querying the smartpen system—without the intervention of your penlet.

In the most common case, only one `PenItem` instance exists for a particular AFD, since the AFD was installed on a particular smartpen and later retrieved from it. Consequently, all data captured by the AFD originated from that smartpen. In the near future, some penlets will have AFDs that can be transferred from one smartpen to another. Such AFDs may have data created by more than one pen. Consider a penlet that allows the annotation of a document by a group of reviewers, each entering comments and then passing the AFD to the next reviewer.

The `PenItem` class encapsulates a smartpen that has penlet data in the ZIP file. It wraps the `AppItems` collection and implements the `IEnumerable` interface.

The `AppItem` class encapsulates a penlet that has data in the ZIP file. Fixed Print applications often have only one penlet associated with a given AFD (although it is possible to build a paper product that is used by multiple Fixed Print penlets). Open Paper applications, however, can write on any Open Paper product, such as a Livescribe notebook.

The `ApplicationName` property is the user-friendly penlet name such as "Paper Replay". The `Path` property is the actual path of the penlet inside the container such as: `userdata\AYE-ABD-FDZ-DJ\Paper Replay` . The `AppItem` class wraps the `InstanceItems` collection and implements the `IEnumerable` interface.

The `InstanceItem` class represents the penlet as a runtime instance on the smartpen. The class has an `InstanceId` property and two very important methods: `GetFileIterator` and `OpenStream`.

To access the penlet data in the ZIP file, iterate over the smartpens that generated penlet data. For each smartpen, iterate over the penlets that stored data in the ZIP file. From each runtime instance, you can access the data inside the ZIP file.

Note: The `PenletData`, `PenItem`, and `AppItem` classes all implement the `IEnumerable` interface defined by the .NET Framework. As a convenience, therefore, you can access the collections they wrap by using the wrapper object as the `in` value of the `foreach` statement.

Penlet vs. Instance

The term ***penlet*** refers to the compiled code associated with a fully-qualified classname. It is the Java byte code stored in a JAR and installed on a smartpen. However, the smartpen system cannot yet execute it without creating some necessary runtime metadata first.

The term ***penlet instance*** refers to compiled code associated with an Instance ID. It is Java byte code stored in a JAR and installed on a smartpen. *And* it is ready to be executed. The difference is that the smartpen system has assigned it an Instance ID to identify the executable as it runs. You can think of the Instance ID like a runtime handle for the penlet.

Potentially, there could be multiple instances of a penlet running simultaneously, each with its own Instance ID.

Each smartpen constitutes a separate "namespace" for its Instance IDs. Thus, within a given smartpen, each Instance ID is unique, but different smartpens can map the same Instance IDs to different penlets.

Instance IDs are assigned dynamically, so events like re-setting the smartpen may cause the Instance IDs to be re-mapped. You need not be concerned about this potential re-mapping, however, since your desktop application will retrieve the Instance IDs from the fully qualified classname each time a smartpen is attached. In other words, in the Penlet Data object model, the AppItem will always return the current InstanceItems collection.

The `AppItem` class encapsulates the penlet directory inside the ZIP container. The `InstanceItem` class encapsulates a particular instance of that penlet: each instance represented by its own directory named after the appropriate Instance ID.

For example, consider a container that has this directory structure :
`userdata\AYE-ABD-FDZ-DJ\Paper Replay\61\...`

`AYE-ABD-FDZ-DJ` identifies the `PenInfo` object

`Paper Replay` identifies the `AppItem` object

`61` identifies the `InstanceItem` object

In general, a container with penlet data has a directory structure like this:
`userdata \ [Pen serial] \ [Penlet name] \ [Instance Id] \` ...

Note: At present, a penlet is executed one instance at a time, so the `InstanceItem` collection for an `AppItem` will have only one element. In the future, you will see penlets where several instances can run simultaneously on a single smartpen. In that case, the `InstanceItem` collection will have more than one element in it.

Application Map

The issue of Instance IDs has a direct impact on static regions. Since static regions are defined before the penlet is installed, the AFD designer has no way of knowing the Instance ID. So, in a static region, the Region ID encodes an Application ID in place of the Instance ID. The Application ID is an arbitrary integer, invented by the AFD designer, that represents the penlet.

The Application Map is a list of key-value pairs, in which the Application ID is the key, and the fully-qualified class name of the penlet is the value. At runtime, a user taps on a static region. The smartpen firmware uses the Application Map to look up the fully-qualified class name of the associated penlet. Then the firmware runs the penlet, creating an Instance ID on the fly.

Accessing Penlet Data Via InstanceItem

The `InstanceItem` class gives you access to the files that contain the penlet data. Let us examine those two important methods:

Since `InstanceItem` class supports `IEnumerable`, we can use `foreach` to iterate the penlet data. Alternatively, you access a file iterator by calling the `GetFileIterator` method.

Using a foreach statement

Following is an example that uses a `foreach` statement to access the files in each `InstanceItem`

```
public bool IteratePenletData(string fileName){
   Container container = new Container(fileName);
   if (!container.IsPenletData()) {
      Console.Out.WriteLine("Not a penlet data file");
      return false;
   }
   foreach (PenItem penItem in container.PenletData) {
      foreach (AppItem appItem in penItem) {
         foreach (InstanceItem instance in appItem.InstanceItems) {
            foreach(LSFile file in instance){
               if (file.Exists()) {
                  if (file.IsDirectory()){
                     Console.Out.WriteLine("[Directory] " + file.GetFileName());
                  } else {
                     Console.Out.WriteLine("[File     ] " + file.GetFileName());
                  }
               }
            }
         }
      }
   }
}
```

Using a File Iterator

The `GetFileIterator` method returns an object for iterating over the files of your penlet data. The penlet can save data in a file hierarchy (that is, a tree of files and directories like a standard file system). In that case, you can iterate over the hierarchy in the familiar way: testing if an item is a file or directory, getting the child items of a directory, retrieving filenames, and reading from or writing to individual files, as desired.

The `OpenStream` method, called with a filename, opens the file and returns a stream.

The following code example assumes that you have a ZIP file containing penlet data and pass it to a method we called `IteratePenletData`. It outputs a listing of directories and files contained in the ZIP file.

```
public bool IteratePenletData(string fileName){
     Container container = new Container(fileName);
     if (!container.IsPenletData()) {
         Console.Out.WriteLine("Not a penlet data file");
         return false;
     }
     foreach (PenItem penItem in container.PenletData) {
         foreach (AppItem appItem in penItem) {
             foreach (InstanceItem instance in appItem.InstanceItems) {

                 FileIterator fileIterator = instance.GetFileIterator();

                 LSFile file = fileIterator.GetNext();

                 while (file != null & file.ptr != IntPtr.Zero) {
                     if (file.Exists()) {
                         if (file.IsDirectory()){
                             Console.Out.WriteLine("[Directory] " +
                               file.GetFileName());
                         }
                         else {
                             Console.Out.WriteLine("[File     ] " +
                               file.GetFileName());
                         }
                     }
                     file = fileIterator.GetNext();
                 }
             }
         }
     }
     return true;
}
```

The file `iterator` returns instances of `LSFile`, which allow you to access the data files inside the ZIP file. The code first tests if the `LSFile` object is a directory, and then writes to the console the string `["Directory"]` or `["File"]` followed by the name of the `LSFile` object.

Penlet Data Retrieval Example

The next sample gets penlet data from the smartpen, saves it in a ZIP file, and accesses the data from the ZIP file. The penlet is a sample called `GetBoungingBoxes`, included in the Desktop SDK. The penlet stores a line of text that describes the bounding box around the last stroke the user wrote on Open Paper. The line of text states the X- and Y-coordinates of the upper-left corner of the bounding box as well as its height and width. That line of text is the penlet data that will be transferred.

You will learn a lot by reading through this sample carefully, so we include the full source code.

```
using System;
using System.Threading;
using System.IO;
using System.Runtime.InteropServices;
using System.Reflection;
using Livescribe.DesktopSDK.PenComm;
using Livescribe.DesktopSDK.PenComm.Interop;
using Livescribe.DesktopSDK.PenData;
using Livescribe.DesktopSDK.AFP;
using PenComm = Livescribe.DesktopSDK.PenComm;

namespace DataCapture {
  class DataCapture
    private static Smartpens smartpens;
    private const string APP_NAME = "DataCapture";
    private const string PENCOMM_LOG = "DataCapture.log";
    private const string PENLET_DATA_FILE = "penletData.zip";
    private const string PENLET_NAME = "GetBoundingBoxes";
    private const string DATA_FILE = "boxes.txt";
    private static bool foundPen = false;
    private static bool handlerRegistered = false;
    private static string tempDir;

    // This sample gets data from the GetBoundingBoxes penlet and displays
    // it on the console.  After displaying the data last captured by the penlet,
    // we establish a data handler and wait for new data to show up.
    static void Main(string[] args) {
       tempDir = Path.GetTempPath();
       System.Console.WriteLine("====================================================");
       System.Console.WriteLine("Read data written by the " + PENLET_NAME + " penlet.");
       System.Console.WriteLine("Data will be stored in:  " + tempDir);
       System.Console.WriteLine("====================================================");
       System.Console.WriteLine("Waiting for pen.");

       InitializePenCommLib();

      // Keep processing until user presses a key to terminate
      while (!Console.KeyAvailable) {
         if (!DataCapture.foundPen) {
            // If we didn't find a pen yet, remind user we're waiting for one.
            Console.Write(".");
         }
         Thread.Sleep(2000);
      }
    }

    // Initialize PenComm library
    private static void InitializePenCommLib() {
       smartpens = new Smartpens(true, PENCOMM_LOG);

       // The Smartpens object keeps track of attach event handlers in
      // PulsePenAttachNormalEvent
      // Create a new callback object for our PenAttachEvent method and
      // register it by adding it to the list of attach event handlers.
      smartpens.SmartpenAttachNormalEvent +=
         new Smartpens.SmartpenChangeCallback(PenAttachEvent);

       // Do the same for our detach event handler.
       smartpens.SmartpenDetachNormalEvent +=
          new Smartpens.SmartpenChangeCallback(PenDetachEvent);

       // Search for any already docked pens
          smartpens.Find();
    }

    // Register our data handler (NewData).  It will be invoked whenever new data
    // arrives from our penlet of interest (PENLET_NAME).
    private static void RegisterForNewApplicationData() {
       if (!handlerRegistered) {
          try  {
             // Application registration information.
             RegAppInfo appInfo = new RegAppInfo();
             appInfo.autoStart = PenAPI.AutoStartTypes.Disabled;
             appInfo.path = Assembly.GetExecutingAssembly().Location;
             appInfo.description = APP_NAME;
             appInfo.version = "1.0";

             // Register Application (that's us)
             AppHandle appHandle = smartpens.DesktopApplications.RegisterApp(appInfo);

             // Register for data callback so that every time the penlet PENLET_NAME has
             // new data, the data will be passed to our callback (NewData).
             PenAPI.DataCallback dataCallback = new PenAPI.DataCallback(NewData);
             smartpens.DesktopApplications.RegisterDataCallback(appHandle, dataCallback,
                null);

             // Register Penlet Information of the GetBoundingBoxes penlet
             RegPackageInfo regPackageInfo = new RegPackageInfo();
             regPackageInfo.appHandle = appHandle;
             regPackageInfo.penId = PenComm.PenId.Zero();
             regPackageInfo.uniqueName = PENLET_NAME;

             // Register the GetBoundingBoxes penlet
             PackageHandle packageHandle =
                smartpens.DesktopApplications.RegisterPackage(regPackageInfo);
             handlerRegistered = true;
          }
          catch (Exception e) {
             HandleError(e);
          }
       }
    }

    private static PenAPI.DataOpTypes NewData(PenDataCallbackInfo dataInfo,
           IntPtr userParam) {
       try {
          // Construct file name that will be unique to a single stroke. We will save our
          // data in that file. We need to save the data in a file in case
          // the stroke is transferred to the desktop app in multiple invocations of the
          // callback.  This may happen, since a stroke can be composed of many
          // co-ordinates.
          String filePath = "Pen" + dataInfo.penId +
               "_Penlet" + dataInfo.penletHandle + "_Time" + dataInfo.startRtc + ".zip";
                System.Console.WriteLine("    NewData: " + (dataInfo.pos+1) + "-" +
                   (dataInfo.pos+dataInfo.bufSize) + " of " + dataInfo.total +
                   " bytes, status = " + dataInfo.packetStatus);
                System.Console.WriteLine("    writing to:  " + filePath);
                filePath = tempDir + filePath;

          // Create or append to the file, depending on whether we're
          // starting with the first co-ordinate
          BinaryWriter bw = null;
             if (dataInfo.pos == 0) {
                bw = new BinaryWriter(new FileStream(filePath, FileMode.Create));
             }
             else {
                bw = new BinaryWriter(new FileStream(filePath, FileMode.Append));
                }

                byte[] buf = new byte[dataInfo.bufSize];
                Marshal.Copy(dataInfo.buffer, buf, 0, dataInfo.bufSize);
                bw.Write(buf);
                bw.Close();

                if (dataInfo.packetStatus != PenAPI.DataPacketStatus.MorePackets) {
                   PenletData penletData = new Container(filePath).PenletData;
                   DisplayData(penletData);
                }
             }
       catch (Exception e) {
          HandleError(e);
       }
       return PenAPI.DataOpTypes.Continue;
```

```
}

// Read and display penlet data.
private static void DisplayData(PenletData penletData) {
   try {
      // The penlet data file is a zip file which is structured like this:
      // userdata\[Pen Serial]\[Penlet Name]\[InstanceId]\...
      // PenItem corresponds to [Pen Serial]
      // AppItem corresponds to [Penlet Name]
      // InstanceItem corresponds to [InstanceId]
      // The PENLET_NAME penlet data file is named:
      //      userdata\[Pen Serial]\[Penlet Name]\[InstanceId]\boxes.txt
      foreach (PenItem penItem in penletData.PenItems) {
         foreach (AppItem appItem in penItem) {
            foreach (InstanceItem instanceItem in appItem.InstanceItems) {
               ContainerStream containerStream = instanceItem.OpenStream(DATA_FILE);
               BinaryReader reader = new BinaryReader(containerStream);
               byte[] data = reader.ReadBytes(1024);
               String value = System.Text.Encoding.ASCII.GetString(data);

               System.Console.WriteLine(value);

               reader.Close();
               containerStream.Close();
            }
         }
      }
      penletData.FileContext.CloseContext();
   }
   catch (Exception e) {
      HandleError(e);
   }
}

// Called when a smartpen is attached.  The first time a pen is attached,
// we check for the penlet and process existing data.  Then we register for
// new data. On subsequent attach events for this smartpen, the attach event handler
// has little effect; however, the NewData callback will be invoked if there is new
// data.
// Note 1: The exception handler for DataGet outputs a warning if the penlet is not
// installed or has not yet been run on the smartpen.
// Note 2: If we don't wish to warn the user that the penlet has not been installed
// or run yet, we can omit the attach event handler altogether, and simply register
// the NewData callback in our InitializePenCommLib method.

private static void PenAttachEvent(Smartpen pen) {
   DataCapture.foundPen = true;
   Console.WriteLine("\n\n--> Pen " + pen.PenSerial + " connected");

   if (!handlerRegistered) {
      PenletData penletData = null;
      try  {
         // Retrieve data for GetBoundingBox penlet and store it in a temporary file.
```

```
            // The data is in zip format and can be processed with any zip reader.
            pen.DataGet(PENLET_NAME, 0, PENLET_DATA_FILE);

            // From the data file we create a container and extract a PenletData object
            // from it for easy processing.
            penletData = new Container(PENLET_DATA_FILE).PenletData;
         }
         catch (Exception e) {
            System.Console.WriteLine ("\nThe " + PENLET_NAME +
               "penlet is not installed on your pen or has not been run successfully.");
            System.Console.WriteLine ("Please install it using the PlatformSDK. ");
            System.Console.WriteLine("Once it is installed, you will need to run the" +
               "penlet and draw a shape on Livescribe Paper.\n");
            HandleError(e);
            return;
         }
         System.Console.WriteLine("Initial Data:");
         DisplayData(penletData);
         RegisterForNewApplicationData();
        }
      }

    static private void PenDetachEvent(Smartpen pen) {
        Console.WriteLine("<-- Pen " + pen.PenSerial + " disconnected");
    }

    private static void HandleError(Exception ex) {
         System.Console.WriteLine("Exception: " + ex.Message, "Error");
         System.Console.WriteLine(ex.StackTrace);    }
  }
}
```

The code relevant to penlet data retrieval starts with the `PenAttachEvent` handler. (The handler is first encapsulated by a delegate of type `SmartPens.SmartpenChangeCallback`, and then the delegate instance is associated with the `SmartpenAttachNormalEvent`.)

```
            // Set callback to retrieve pen attach events
            smartPens.SmartpenAttachNormalEvent += new
               Smartpens.SmartpenChangeCallback(PenAttachEvent);
```

When the user attaches a smartpen to the computer, the PenComm Service raises this event, which calls the `PenAttachEvent` handler.

The `PenAttachEvent` method starts by getting penlet data from the `PENLET_NAME` penlet (a sample penlet called `GetBoundingBoxes`) and transfers it to the `PENLET_DATA_FILE` file on the desktop computer's file system. The `PENLET_DATA_FILE` is arbitrarily named `penletData.zip`.

```
                pen.DataGet(PENLET_NAME, 0, PENLET_DATA_FILE);
```

The handler creates a `PenletData` object hierarchy by instantiating a `Container` and passing it the ZIP file, which is `PENLET_DATA_FILE`.

```
penletData = new Container(PENLET_DATA_FILE).PenletData
```

Then the handler calls our `DisplayData` method, passing the `penletData` object. Finally, it calls our `RegisterForNewApplicationData` method in order to register the penlet and the data callback.

The custom `DisplayData` method uses the familiar nested `foreach` statements to iterate over the `PenletData` hierarchy. The penlet data was obtained by calling `pen.DataGet` inside the `PenAttachEvent` handler. Consequently, there is only one smartpen—the `Smartpen` object returned by the PenComm Service in the pen parameter of the `PenAttachEvent` method. It is represented by a single `SmartpenItem` object. There is also only one penlet, identified by the `PENLET_NAME` passed to the `pen.DataGet` method. It is represented by a single `AppItem`.

The `InstanceItem` object is the single element in its collection. The `DisplayData` method calls `OpenStream` on that object and passes the `DATA_FILE`. This is a file called `boxes.txt`, contained in the ZIP file. The developer of this sample was familiar with the `GetBoundingBoxes` penlet and knew that it stores a single file called `boxes.txt`. The code does not bother to call the `GetFileIterator` method on the `InstanceItem`, since no file hierarchy exists in this penlet data from this penlet.

An important fact is worth stating here. The contents of the penlet storage (filenames, file hierarchies, and data types in the files) are up to the penlet developer. It follows that the desktop application developer must be completely familiar with how the penlet structured and stored the penlet data.

Note: Desktop applications may wish to access audio files and sessions that were recorded with Livescribe's Paper Replay. To assist you, the Desktop SDK provides classes that simplify reading the file format structure and file formats of Paper Replay. Thus, you do not need in-depth knowledge of how Paper Replay audio and sessions are stored.

The `OpenStream` method returns an `ContainerStream` object, which must be passed to a stream reader or writer. In this case, we are reading a file from the computer file system (the `boxes.txt` in the ZIP file). So, we create a `BinaryReader` with the opened stream. Then we read 1024 bytes from the data.txt file, convert them to a `String`, and write it to the console.

After the first attach event for the smartpen, the smartpen event handler will do very little. Notice the `if` statement in the smartpen attach event handler:

```
if (!handlerRegistered) {

//  MOST OF THE HANDLER'S FUNCTIONALITY IS IN THESE BRACES!

}
```

The `NewData` callback has been registered and will handle data transfers on subsequent attach events.

Lastly, we should note that the `NewData` callback gets the transferred data from the `dataInfo` parameter that the PenComm Service passes to your application. Each callback invocation holds up to 1008 bytes of data in `dataInfo.buffer`.

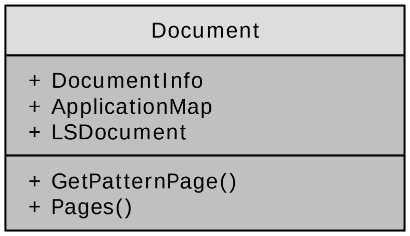
```
BinaryWriter bw = null;
   if (dataInfo.pos == 0) {
      bw = new BinaryWriter(new FileStream(filePath, FileMode.Create));
   }
   else {
      bw = new BinaryWriter(new FileStream(filePath, FileMode.Append));
      }

      byte[] buf = new byte[dataInfo.bufSize];
      Marshal.Copy(dataInfo.buffer, buf, 0, dataInfo.bufSize);
      bw.Write(buf);
      bw.Close().
```

The callback appends the data to a temporary file because penlet data can be longer than 1008 bytes, which requires multiple invocation . In this particular case, though, our single line of text is less than 1008 bytes.

The `NewData` callback finishes up much like the smartpen attach event handler. It creates a `Container` object, gets the `PenletData` object, and passes it to the `DisplayData` call.

Accessing an AFD

Now let us consider how to proceed when the contents of the ZIP file are an AFD, instead of penlet data. As a reminder, a very high-level view of the Document class look likes this. Many methods and properties are omitted:



The `DocumentInfo` property can be used to get the metadata of an AFD. The following full-fledged example outputs a list of all packages—both penlets and AFDS—on the smartpen. We will concentrate on the AFD portion of the sample. The `PenAttachEvent` method tests whether the package is an AFD, and then calls our custom `PrintReadAFDInfo` method.

```
using System;
using System.IO;
using System.Threading;
using Livescribe.DesktopSDK.PenComm;
using Livescribe.DesktopSDK.PenComm.Interop;
using Livescribe.DesktopSDK.PenData;
using Livescribe.DesktopSDK.AFP;

namespace ListPackages
{
    class ListPackages {
        private static Smartpens smartpens;
        private const string PENCOMM_LOG = "ListPackages.log";
        private const string DATA_FILE = "data.zip";
        private static bool foundPen = false;
        private static bool complete = false;

        static void Main(string[] args) {
            System.Console.WriteLine("================================================");
            System.Console.WriteLine("List packages (penlet or paper product).");
            System.Console.WriteLine("Display AFD information and penlet data");
            System.Console.WriteLine("================================================");
            System.Console.WriteLine("Waiting for pen.");

            InitializePenCommLib();

            // Windows programming should avoid ending while a callback is busy.
            // Also we should give user some indication that we're still waiting,
            //if we haven't found a pen yet.
            while (!complete) {
                if (!foundPen) {
                    System.Console.Write(".");
                }
                Thread.Sleep(2000);
            }
        }

        // Initialize PenComm library
        private static void InitializePenCommLib() {
            smartpens = new Smartpens(true, PENCOMM_LOG);

           // The Smartpens object keeps track of attach event handlers in
           // SmartpenAttachNormalEvent
           // Create a new callback object for our PenAttachEvent method and
           // register it by adding it to the list of attach event handlers.
           smartpens.SmartpenAttachNormalEvent +=
              new Smartpens.SmartpenChangeCallback(PenAttachEvent);

           // Search for any already docked pens
           smartpens.Find();
        }

        // This is called when a pen is attached
        private static void PenAttachEvent(Smartpen pen) {
            foundPen = true;
            try {
                // Refresh packages list
                pen.Packages.Update();

                // Show package data
                foreach (PackageItem packageItem in pen.Packages.Items) {
                    System.Console.WriteLine ("\n==============================");
                    System.Console.WriteLine ("\nPackage: " + packageItem.PackageName +
                        " (" + packageItem.FullPath + ")" );

                    // Retrieve the data of the package
                    pen.DataGet (packageItem.PackageName, 0, DATA_FILE);

                    // Anything that comes from the pen can be opened as a container
                    try {
                        Container container = new Container(DATA_FILE);

                        // If this container is a document, we can access
                        // the document member
                        if (container.IsDocument()) {
                            PrintReadAFDInfo(container.Document);
                        }

                        // If this container is a penlet, we can access
                        // the penlet data member
                        if (container.IsPenletData()) {
                            PrintPenletData(container.PenletData);
                        }

                        container.Close();
                    }
                    catch (Exception ex) {
                        System.Console.WriteLine("Exception: " + ex.Message, "Error");
                    }

                    File.Delete(DATA_FILE);
                }
            }
            catch (Exception ex) {
                System.Console.WriteLine("Exception: " + ex.Message, "Error");
            }
            finally {
                complete = true;
            }
        }

        // Print AFD information
        private static void PrintReadAFDInfo(Document document) {
            string afdVersion;
            if (document.DocumentInfo.GetAFDVersion(out afdVersion)) {
                Console.Out.WriteLine("AFD Version: " + afdVersion);
            }

            string guid;
            if (document.DocumentInfo.GetGUID(out guid)) {
                Console.Out.WriteLine("GUID: " + guid);
            }

            string author;
            if (document.DocumentInfo.GetAuthor(out author)) {
                Console.Out.WriteLine("Author: " + author);
            }

            Console.Out.WriteLine("Number of pages: " + document.GetNumberOfPages());
        }

        // Print the penlet data files
        private static void PrintPenletData(PenletData penletData) {
            foreach (PenItem penItem in penletData) {
                foreach (AppItem appItem in penItem) {
                    foreach (InstanceItem instance in appItem.InstanceItems) {
                        foreach(LSFile file in instance) {
                            PrintFile(file, "");
                        }
                    }
                }
            }
        }
```

```
        // Print the director/file structure.
        private static void PrintFile(LSFile file, string spacing) {
            if (file.Exists()) {
                Console.Write(spacing);
                if (file.IsDirectory()) {
                    Console.Out.WriteLine("[Directory] " + file.GetFileName());
                    foreach (LSFile f in file) {
                        PrintFile(f, spacing + "  ");
                    }
                }
                else {
                    Console.Out.WriteLine("[File     ] " + file.GetFileName());
                }
            }
        }
    }
}
```

The `PrintReadAFDInfo` method takes a `Document` object. It creates a `Container` object, using the filename. It then tests whether the contents of the ZIP file is truly an AFD by calling the `IsDocument` method. If it is an AFD, then the code gets the `Document` property and proceeds to get the AFD's version, Guid, author and number of pages.

Pages in an AFD

You are probably looking to retrieve something more exciting from your AFD than just the metadata. You may be interested in retrieving the stroke data! Before jumping into that stimulating topic, however, let's examine the nature of the pages in an AFD. Although not complicated, this issue has several ramifications that we should explore.

An AFD affects pages in three ways:

- Contains the pages that make up a paper product.
- Defines the composition and layout of pages.
- Contains the pattern for each page.

Pattern: Pages and Copies

Let us assume that you are defining a paper product—for example, a short notebook. You probably wish to produce multiple copies of that notebook. If all copies have the same pattern, then your user can use her first notebook with no problem. If she starts writing in a second notebook, however, you smartpen will confuse the strokes with strokes from the first notebook. How should you handle this situation? You have three possible strategies: Retirement, Series, and Completely Unique Pattern.

Retirement

With this strategy, once your user fills up the notebooks, she runs your desktop application and "retires" all the strokes and penlet data. You can choose to archive the penlet data for her on the desktop computer. Then you can remove all strokes and penlet data from the AFD. Now, the user can buy another of your notebooks and keep writing. Only one notebook will be active at a time.

Series

With this strategy, you create multiple series of your notebook—for instance, Series A, Series B, Series C, etc. Each series has pattern distinct from the other series. But within a series, multiple notebooks all have the same pattern. Thus, page 5 of a Series A differs from page 5 of a Series B or a Series C notebook. But page 5 of every Series A notebook has identical pattern.

On the Livescribe Smartpen Platform, a "series" can be represented by the concept of *copy*. The values of a Copy property in the Desktop SDK APIs is an unsigned integer. So, an AFD page specified as Page 5, Copy 0 corresponds to page 5, Series A in the preceding paragraph. And Page 5, Copy 1 corresponds to Page 5, Series B. And so forth. Note that Pages begin at 1 and Copies begin at 0.

The entity determined by a Page number and a Copy number is technically called a *page instance*. We refer to them more casually as *pages*, when no confusion would ensue. For the sake of clarity, you will sometimes see *page instance* in this manual.

Absolutely Unique Pattern

With this strategy, every copy of a notebook has unique pattern. There are no series. Each page of each notebook you produce has absolutely unique pattern. The Copy value identifies each notebook uniquely within its AFD. It serves to distinguish all Page 5's from each other, and all Page 6's, etc. Obviously, your Copy values will be as large as the number of individual notebooks you produce. This strategy uses *a lot* of pattern.

Page Templates

Within a paper product definition, the layout of graphic images on each page may not be unique. If you're producing a small notebook, you will probably have just a left and a right page layout. In that case, you can create a right page template and a left page template and apply them throughout the notebook. There's no need to design a separate page layout for each page instance. Conceptually, a page template is like a Master Page in Adobe Framemaker and similar desktop publishing products.

Page templates are not a concern when you are retrieving AFDs from a smartpen. However, they prove very useful if you decide to create an AFD programmatically from your desktop application.

Retrieving Strokes

If your ZIP file contains an AFD, you can access data for any strokes that users may have written on the pages. The data describes the timing and geometry of the strokes.

Your general approach should be as follows:

1. Get the AFD from the smartpen.
2. Create a `Container` object, and verify that it is a document (an AFD, not penlet data), and then get the `Document` property.
3. Call the `Document.GetPatternPage` method, passing the `Page` and `Copy` values for each page instance. The method returns the appropriate PatternPage object.
4. In the `PatternPage` object, get the Pens property, which returns a collection of `StrokeOwner` objects. `StrokeOwner` is a class with a very specific purpose, which we explain below. It is not the same as an attached smartpen, which is represented by an object of the `Smartpen` class.
5. For each `StrokeOwner` object, get the StrokeCollection object.
6. For each key-value pair in the StrokeCollection object, get the key (the time) or the value (a `Stroke` object).

The `PatternPage` class encapsulates the pattern page that is assigned to a page instance. It is a very simple class that wraps a dictionary whose values are `StrokeOwner` objects. It also implements the `IEnumerable` interface, so you can iterate over the `StrokeOwner` objects with a `foreach` statement like this.

```
foreach (StrokeOwner strokeOwner in patternPage)
   {
       // Process each StrokeOwner object here
   }
```

The `StrokeOwner` class represents all the strokes on a pattern page that belong to a particular smartpen. That is its whole purpose. Multiple smartpens may have written on a page. This situation can occur if several smartpen users have your penlet installed. It can also occur if your AFD has Open Paper sections, on which any smartpen could write, whether it has your penlet installed or not.

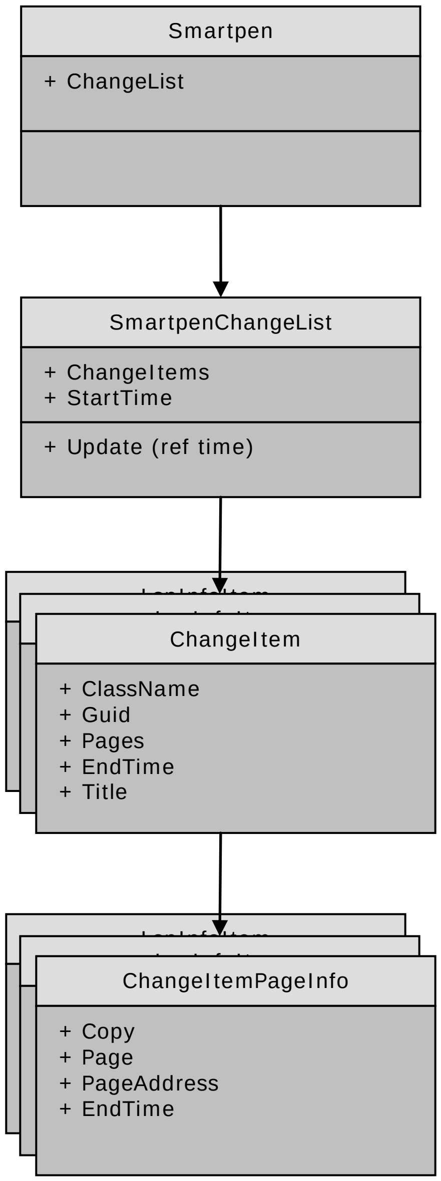
The `StrokeOwner` class derives from the `SmartPenInfo` class and inherits the `SmartPenId` and `SmartPenSerial` properties. It wraps a strokes collection, returned by the Strokes property.

The `StrokeOwner` class derives from the `PenInfo` class, which has the `PenId` property. From the `PenId` property, we can access the pen serial using the `ToString` method. For example: `strokeOwner.PenId.ToString()`.

Using the Smartpen Change List

The approach to stroke retrieval discussed so far requires you to iterate over every page in an AFD. You can potentially retrieve a large amount of data strokes every time. In the interest of data efficiency, you may wish to implement an incremental approach to stroke retrieval. One means to do so is the smartpen change list.

Following is a very basic diagram of the Smartpen Change List hierarchy. Please note that many methods and properties have been omitted:



The `Smartpen` class encapsulates a smartpen attached to the desktop computer. Its `ChangeList` property returns the `SmartpenChangeList` object for the smartpen.

The `SmartpenChangeList` class encapsulates a change list for the current smartpen. The `StartTime` property, which returns the time at which the change list began to search for changes to penlet data and AFDs. It wraps the `ChangeItems` collection, which is returned by the `ChangeItems` property.

To obtain a change list from a specific time, call the `SmartpenChangeList.Update` method and pass a time after which you wish to obtain all changes on the smartpen. This parameter establishes the value of the StartTime property of the `SmartpenChangeList` object.

The relevant code snippet looks like this:

```
ulong time = 0;// from beginning
SmartpenChangeList changeList = pen.ChangeList;
changeList.Update(time);
```

The `ChangeItem` class encapsulates an individual change list item. If the change item is an AFD, it the Pages property returns the collection of `PageChangeItem` objects. The PageChangeItem class represents a page instance that has new strokes on it. It contains the following properties:

Member	**Description**
Copy	Useful mainly for passing to the **`Document.GetPatternPage`** method.
Page	Also passed to the **`Document.GetPatternPage`** method. Together the **`Copy`** and **`Page`** values determine a page instance.
PageAddress	The unique address of a pattern page. A page address is a 64-bit unsigned integer. Each byte represents one portion of the address: SEGMENT.SHELF.BOOK.PAGE. Portions of the address are separated by dots. Example: **`12.10.7.8`**
EndTime	Time of the last stroke on the page instance.
LSDocument	Low-level object for manipulating documents. The LSDocument class is defined in the AFP API.

Stroke Retrieval Example

The following code sample makes use of a smartpen change list when getting an AFD from the smartpen and then reading stroke information from the AFD. You may wish to study it carefully.

The sample "decodes" strokes in the sense that for each stroke, it gets an array of points. Each point represents the co-ordinates of a vertex of a geometric shape that abstracts the stroke.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Threading;
using Livescribe.DesktopSDK.PenComm;
using Livescribe.DesktopSDK.PenComm.Interop;
using Livescribe.DesktopSDK.PenData;
using Livescribe.DesktopSDK.AFP;

namespace DecodeStrokesForAllAFDs {
    class DecodeStrokesForAllAFDs {
        private static Smartpens smartpens;
        private const string PENCOMM_LOG = "DecodeStrokesForAllAFDs.log";
        private const string DOC_DATA_FILE = "data.zip";
        private static string STROKE_FILE = Environment.CurrentDirectory +
          "\\strokes.txt";
        private static bool foundPen = false;
        private static bool complete = false;
        private static bool errorsOccurred = false;

        // This sample gets strokes from pen and writes them to stroke.txt
        static void Main(string[] args) {
            System.Console.WriteLine("==============================================");
            System.Console.WriteLine("Read strokes written on all paper products" +
               "for a single pen");
            System.Console.WriteLine("Decoded strokes are written to:" + STROKE_FILE);
            System.Console.WriteLine("==============================================");
            System.Console.WriteLine("Waiting for pen.");

            InitializePenCommLib();

            // In Windows, we should avoid ending while a callback is busy.  Also, if we
            // haven't found a pen yet, we should give tell user that we're still
waiting.
            while (!complete) {
                if (!foundPen) {
                    System.Console.Write(".");
                }
                Thread.Sleep(2000);
            }

            Console.Out.WriteLine("\n<Press a key to continue>");
            Console.ReadLine();
        }

        // Initialize PenComm library
        private static void InitializePenCommLib() {
            smartpens = new Smartpens(true, PENCOMM_LOG);
            // The Smartpens object keeps track of attach event handlers in
            // PulsePenAttachNormalEvent
            // Create a new callback object for our PenAttachEvent method and register it
            // by adding it to the list of attach event handlers.
            smartpens.SmartpenAttachNormalEvent
               += new Smartpens.SmartpenChangeCallback(PenAttachEvent);

            // Search for any already docked pens
            smartpens.Find();
        }

        // Called when a pen is attached
        private static void PenAttachEvent(Smartpen pen) {
            foundPen = true;
            TextWriter strokeFile = new StreamWriter(STROKE_FILE);
            try {
                System.Console.WriteLine("\nFound Pen: " + pen.PenSerial);
                ulong time = 0; // from beginning
                System.Console.WriteLine("Read change list from pen from time: " + time);

                SmartpenChangeList changeList = pen.ChangeList;
                // Update the change list (which started out empty) so that it contains
                // all the changes since time.
                changeList.Update(time);

                // The changeList contains information (such as end time of the change,
                // notebook guid, unique package name) of data changes made on
                // pen (ChangeItems). Each ChangeItem describes change information for a
                // paper product or penlet.
                System.Console.WriteLine("Decoding paper product:");
                foreach (ChangeItem item in changeList.ChangeItems){
                    // If the item change information of a paper product.
                    if (!String.IsNullOrEmpty(item.Guid)) {
                        DecodeStrokes(pen, item, time, strokeFile);
                        if (errorsOccurred)
                            break;
                    }
                }
            }
            catch (Exception e) {
                HandleError(e);
            }
            finally {
                strokeFile.Close();
                complete = true;
                System.Console.WriteLine("\nComplete:  " + (errorsOccurred ? "not " : "")
                   + "all paper products processed.");
            }
        }
```

```
// Decode all the strokes for a particular paper product and write them to a file
private static void DecodeStrokes(Smartpen pen, ChangeItem item, ulong time,
   TextWriter strokeFile) {
    strokeFile.WriteLine("\n========================================");
    strokeFile.WriteLine(String.Format("Document: {0}", item.Title));
    System.Console.WriteLine("   " + item.Title);

    // We will extract strokes from pen and store them in a temporary file whose
    // name is based on the paper product name.
    string fileName = String.Format("{0}_{1}", item.Title, DOC_DATA_FILE);
    try {
        // Pen stores strokes in files on pen's flash memory system.
        // The pen uses one file for each paper product: name of the file
        // is the GUID of the paper product.  We retrieve the stroke data and
        // store it locally in a temporary file.
        pen.DataGet(item.Guid, 0, fileName);

        // From the data file we create a container and extract a document object
        // from it for easy processing
        Document doc = new Container(fileName).Document;

        // Strokes are organized by pages.  Decode strokes for each page.
        foreach (PageChangeItem pageInfo in item.Pages) {
            strokeFile.WriteLine("\n-----------------------------");
            strokeFile.WriteLine(String.Format("Copy: {0}, Page: {1}",
               pageInfo.Copy, pageInfo.Page));

            // For each page, we find its unique PatternPage object.
            // The PatternPage is a collection of strokes created by pens.
            PatternPage patternPage = doc.GetPatternPage(pageInfo.Page,
               pageInfo.Copy);

            // Each element of the PatternPage is a StrokeOwner object which
            // contains pen information (such as PenSerial) and strokes created
            // by that pen
            foreach (StrokeOwner penData in patternPage) {
                foreach (KeyValuePair<long, Stroke> keyPair in penData) {
                    // Get time at which the stroke was created
                    long creationTime = keyPair.Key;
                    strokeFile.WriteLine("\nTime ({0}): ", creationTime);

                    // A single stroke can contain many points.
                    // The pen records up to 70 points per second.
                    Stroke stroke = keyPair.Value;
                    Shape shape = stroke;
                    System.Drawing.Point[] points = shape.GetVertices();
                    foreach (System.Drawing.Point point in points) {
                        // Write the point of strokes to file.
                        // Upper left corner of page is 0,0 with the y coordinate
                        // increasing down the page. The unit of measurement is
                        // Anoto Unit (au), which are roughly 677 dpi.
                        strokeFile.Write(String.Format(" ({0},{1})", point.X,
                        point.Y));
                        }
                    }
                }
            }
            doc.Close();
        }
        catch (Exception e) {
            HandleError(e);
        }
        finally {
            File.Delete(fileName);
        }
    }

    private static void HandleError(Exception ex) {
        errorsOccurred = true;
        System.Console.WriteLine("Exception: " + ex.Message, "Error");
    }

}
}
```

This sample does most of the things we talked about in the preceding sections.

Since we want to retrieve stroke information from the smartpen, we are interested in AFDs. In the `PenAttachEvent` method, we test each `ChangeItem` to see if it is an AFD by determining if the `Guid` property is null or the empty string. The code snippet is:

```
foreach (ChangeItem item in changeList.ChangeItems) {
    if (!String.IsNullOrEmpty(item.Guid)) {
        DecodeStrokes(pen, item, time, strokeFile);
```

The `strokeFile` is the ultimate output of this sample—a file where we will store the vertex information for all strokes retrieved from the AFD.

The `DecodeStrokes` method is called with each `ChangeItem` that is an AFD. The method body creates an appropriate ZIP filename for each AFD that we request from the smartpen. Each filename incorporates the AFD's name (`item.Title`).

```
string fileName = String.Format("{0}_{1}", item.Title, DOC_DATA_FILE);
try {
    // Get data file from pen
    pen.DataGet(item.Guid, 0, fileName);
```

In your own code, you could further distinguish ZIP files by concatenating the following:

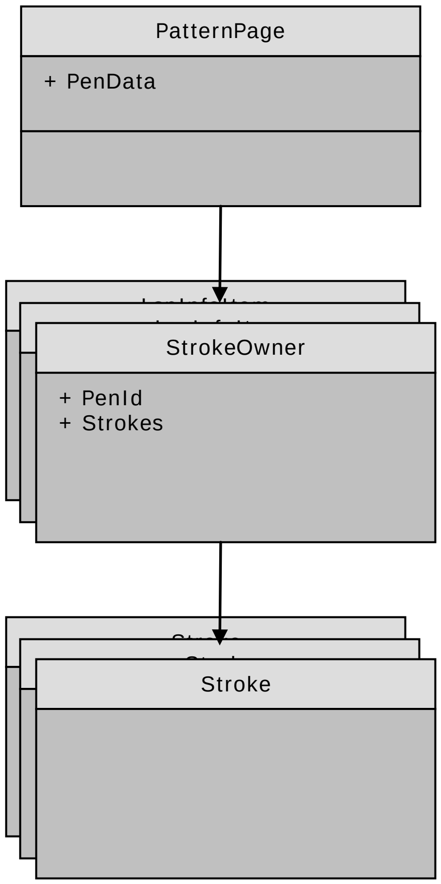
- the GUID string of the AFD. The `ChangeItem.Guid` property returns this value.
- the time of the change list item. The `ChangeItem.EndTime` property returns this value.

The `DecodeStrokes` method then calls the `pen.DataGet` method, passing the GUID of the AFD and the filename where the transferred strokes should be stored.

Since strokes are retrieved from a `PatternPage` object, we would like to call the `Document.GetPatternPage` method. However, that method takes a `page` and `copy` parameter. We must first get these from the `Page` and `Copy` property of the `ChangeItemPageInfo` object. The collection of `PageChangeItem` objects for the AFD is returned by the `Pages` property of the `ChangeItem object`. The following code snippet puts everything together:

```
foreach (PageChangeItem pageInfo in item.Pages) {
    strokeFile.WriteLine("\n------------------------------");
    strokeFile.WriteLine(String.Format("Copy: {0}, Page: {1}",
       pageInfo.Copy, pageInfo.Page));
    PatternPage pp = doc.GetPatternPage(pageInfo.Page, pageInfo.Copy);
```

Now we get the strokes on the pattern page. In this discussion, a high-level diagram of the objects involved in retrieving strokes might be helpful. Please note that many methods and properties are omitted.



Strokes are organized into a collection for each smartpen that wrote on the page. The `StrokeOwner` class wraps a collection of `Stroke` objects. The `PatternPage` object wraps a collection of StrokeOwner objects, and the `PenData` property returns that collection.

So, for each `StrokeOwner` object in the `PatternPage`, we access the dictionary of strokes (where the key is the time and the value is the Stroke). For each key-value pair in the dictionary, we get the key and output it to the console as "Time (*key*):". Then we get the value and assign it to a `Stroke` object.

Since a stroke can be considered a shape, we assign the `Stroke` object to a `Shape` variable named `shape`. Because of the implicit cast in the Stroke class, an appropriate `Shape` is automatically created and a reference assigned to `shape`. (C#'s implicit casts can be powerful!)

```
// Each element of the PatternPage is a StrokeOwner object which
// contains pen information (such as PenSerial) and strokes created
// by that pen
```

```
foreach (StrokeOwner penData in patternPage) {
    foreach (KeyValuePair<long, Stroke> keyPair in penData) {
        // Get time at which the stroke was created
        long creationTime = keyPair.Key;
        strokeFile.WriteLine("\nTime ({0}): ", creationTime);

        // A single stroke is comprised of potentially many points.
        // The pen records up to 70 points per second.
        Stroke stroke = keyPair.Value;
        Shape shape = stroke;
        System.Drawing.Point[] points = shape.GetVertices();
        foreach (System.Drawing.Point point in points) {
            // Write the point of strokes to file.
            // Upper left corner of page is 0,0 with the y coordinate
            // increasing down the page. The unit of measurement is
            // Anoto Unit (au), which are roughly 677 dpi.
            strokeFile.Write(String.Format(" ({0},{1})", point.X,
            point.Y));
        }
```

.

Once we have a `Shape` object, we call **`GetVertices`** on it. The call returns an array of C# `System.Drawing.Point` objects. Then we output the X- and Y-coordinate for each Point object.

Note: Co-ordinates are in Anoto Units, which are approximately 677 dots per inch. The origin is at the top-left corner of the page

The `Shape` class is in the AFP Wrapper API.

The AFP Wrapper API

The AFP API is a C# wrapper of the Anoto Functionality Platform, which provides low-level Print and Document calls for your AFP desktop application. The AFP API provides all the functionality you need, and you can create desktop applications using only this layer of the Desktop SDK. The Pen Data API and Rendering API, however, provide convenience classes that can greatly simplify your work. In practice, your desktop applications will be a mixture of calls into the Pen Data and Rendering APIs and calls into the AFP API.

Not all low-level classes and functions are mirrored in the higher-level APIs. Our guiding principle was to create a Pen Data or Rendering API class if it could add value by simplifying complexity. In cases where no simplification is possible, you will use the AFP API classes directly.

This section presents the most important AFP classes, including those which provide:

- more powerful functionality than classes in the Pen Data and Rendering API
- basic functionality required by almost any desktop application.

File and Document Access

These classes provide access to AFDs on your desktop computer and the files within them. They also provide ways to merge AFDs.

LSDocument

The `LSDocument` class provides low-level access and manipulation of AFDs on your desktop computer. The corresponding class in the PenData API is `Document`.

The two classes differ in how they cause AFDs to be loaded into memory on your computer. The `Document` class loads *all* pages of the AFD into memory. The LSDocument class employs on-demand loading, by which pages are loaded into memory as they are requested for data access or page rendering. For improved performance and memory footprint, use `LSDocument`.

Merging AFDs

Desktop applications generally request smartpen data incrementally—for instance, using `Smartpen.ChangeList`. To operate on the smartpen data increments, you will probably wish to merge them into a single AFD. The `LSDocument.MergeDocument` method enables you to perform merges.

Note: In order for your smartpen data files to merge in an intuitive and convenient manner, you should choose filenames that indicate smartpen ID, penlet name, and date of retrieval. Merging becomes a simple process of doing the following, according to the nature of your data:

- adding new files to a directory tree in the merged AFD (in situations where older data must be preserved).
- replacing files with identical names (in situations where new data replaces old data).

LSFile

The `LSFile` class permits your desktop application to access individual files, either inside an AFD or on the file system of your computer. The constructor requires the file path in the AFD or on the file system. Useful methods include: `LSFile.Copy`, `LSFile.Delete`, and `LSFile.Move`. The Pen Data API contains no corresponding class.

ContainerStream

The `ContainerStream` class creates a C# stream. You can then pass the stream to a reader or writer for reading from or writing to an AFD.

Consult C# documentation from Microsoft if you need more information on C# streams.

Pages in an AFD

Your desktop application can access the pages in an AFD, using the `PageTemplate`, `PageInstance`, and `PageAddress` classes in the AFP API. As explained above, a *page template* describes a page's characteristics and contains only static information for the page: the height and width dimensions, graphic images, and static regions. A *page instance* represents a physical page: it contains unique dot pattern and dynamic information such as strokes written with a smartpen and data created by penlets.

PageTemplate

In many paper products, such as a notebook, multiple page instances share the same page template. The high-level `Document.Pages` property returns a collection which indicates the template for every page. Several pages can share the same template. The graphic images and static regions are identical for all pages that have the same template.

In a standard notebook, for instance, there is a right-hand page template and a left-hand page template. The `Document.Pages` property returns a collection of `Page` objects, each of which contains references to the appropriate `PageTemplate` object. Thus, pages 2, 4, 6 all point to the same `PageTemplate` object. And pages 1, 3, 5 all point to the same `PageTemplate` object.

The `Page` class in the Pen Data API corresponds to the `PageTemplate` class in the AFP API.

PageInstance

You access the strokes and dynamic regions of a page through the page instance. In the Pen Data API, `Document.GetPatternPage` returns the appropriate `PatternPage` object, through which you can access strokes, for example.

The `PatternPage` class in the Pen Data API corresponds to the `PageInstance` class in the AFP API. Use PageInstance class if you need functionality not provided in the higher-level class. The PageInstance constructor takes a `Copy` and a `Page` parameter, which are the same as for the `Document.PatternPage` method.

A page instance is determined by a unique combination of `Page` and `Copy` values. Note that page instances with the *same* `Page` value and *different* `Copy` values have the same page template but different dot pattern. For example: Page 5, Copy 0 and Page 5, Copy 1 have the same graphic images and static regions. However, once a smartpen user has written on them, they contain different strokes and dynamic regions (if any). See Pattern: Pages and Copies.

Graphics Collection

Each page template contains a collection of graphic objects used by all its page instances. The `Page` object has a `RegionCollection` property that returns the

GraphicsCollection object for the page. A GraphicCollection is a collection of GfxElements.

The Page object has a RegionCollection property and a GraphicsCollection property. RegionCollection contains a collection of active areas. You can iterate by using statement like this:

```
foreach (KeyValuePair<RegionId, Shape> kvp in RegionCollection)
```

GraphicsCollection contains a collection of graphical objects (such as images). You can iterate by using statement like this:

```
foreach (KeyValuePair<GraphicsId,GfxElement> kvp in page.GraphicsCollection)
```

Actual elements in the GraphicsCollection are GfxImage and GfxDrawingArea objects. Both the GfxImage and GfxDrawingArea classes, however, have an implicit cast to GfxElement.

A graphic on a page template is uniquely identified by a 64-bit unsigned integer. Part of the Graphics ID encodes a z-order for layering of graphics on a page. Fifteen bits of the Graphics ID are dedicated to the z-order.

One bit of the Graphics ID encodes whether the image is a control. Images that are identified as controls, can be turned on or off *as a group* by setting the boolean DrawControls property of the Renderer object. You can check if an image is a control or not by setting the Boolean isControl property of the GraphicsId object associated with the image.

Direction of the GraphicsCollection Z-Order

The z-order of the graphics collection applies to layers of images on a page in a first-come, first-painted order. In other words, the lowest z-order layer is painted first, and the highest z-order layer is painted last.

Consequently, the lower z-order layers are conceived of as "farther" from the human viewer of a screen or printed page. And the higher z-order layers are "closer" to the human viewer. See Direction of the RegionCollection Z-Order.

Graphics in a paper product must be in the EPS format. The GfxImage class represents a graphic image on the page. It is associated with an actual graphics file on the desktop computer file system. You need to create objects of this class when rendering an AFD page to the computer monitor. The general procedure is as follows:

1. Create a `GfxImage` object and add it to the graphics collection of the `Page` object.
2. For each distinct graphic image, add the associated EPS file to your desktop application's resources.
3. Add a bitmap version of the EPS file, if your desktop application is rendering pages to the monitor (for example, in a simple page viewer). They are needed to render the background images.

Static Region Collection

Each page template has a collection of static regions, called `RegionCollection`. Static regions are invisible to the smartpen user but are required for the smartpen to perform an action when the user interacts with that portion of the page. For example, tapping the smartpen on a region may play an audio file. Static regions have a location on the page just like a graphic image. Note that a graphic image usually occupies the same location as a static region, indicating to smartpen users which portions of the page are "active."

A static region is uniquely identified by a 64-bit unsigned integer. The bits of the Region ID have various meanings, which are explained in the Region ID page of the Javadoc for the Livescribe Smartpen Platform in the Livescribe Platform SDK. The most important portion is the 16 bits containing the Area ID, which provides the connection between a static region and functionality in the smartpen. Several static regions can contain the same Area ID, if the penlet should perform the same functionality for all of them.

Direction of the RegionCollection Z-Order

Part of the Region ID encodes the z-order of regions that overlap on the same location on a page.

The z-order of a `RegionCollection` applies to layers of regions directly opposite to the z-order of `GraphicsCollection`. Consequently, the lower z-order layers are conceived of as "closer" to the human viewer of a screen or printed page. And the higher z-order layers are "farther" from the human viewer. See Direction of the GraphicsCollection Z-Order

Stroke Collections and Strokes as Shapes

The `StrokeOwner` class in the Pen Data API has a `Strokes` property, which returns the collection of strokes created by a given smartpen on the current page instance.

Strokes are represented by the `Stroke` class in the AFP API. A stroke can be represented as a geometric figure—either a polygon or polyline. A stroke has a creation time and an X,Y coordinate pair for each vertex of the corresponding polygon or polyline.

If you wish to retrieve the vertices of the stroke's shape, simply assign the `Stroke` object to a `Shape` variable. The implicit cast in the `Stroke` class makes this possible.

The shape of a stroke is significant if you wish to perform your own handwriting recognition on strokes or if you wish to render strokes to the computer monitor.

Intersecting Strokes and Regions

A common task for desktop applications involves determining whether a given region has strokes in it. For example, a region that acts as a check box would contain strokes if the smartpen user selected it by writing anywhere in the region.

To determine if a region has strokes, use the `RegionCollection`, `StrokeCollection`, and intersection. The key concept to keep is mind is that regions and strokes are both geometric shapes. In fact, both are derived from the `Shape` class. If strokes were made in a region, then they overlap the same location on the page. The AFP API provides methods to compute intersections of region and stroke shapes. If the intersection succeeds, then you know that strokes were made in the region.

The general procedure is as follows:

1. Get a stroke collection for a page instance (a `PatternPage` object). See Retrieving Strokes.
2. Get the appropriate `page template` (a `Page` object) associated with the current pa.
3. Get the `RegionCollection` for that `PageTemplate` object

Note: You must know the structure of the page template and the Region ID of the region you are interested in. Either you are the designer of the AFD, or you must communicate with the designer and discover what the different regions are designed to do. Access to the penlet code is extremely useful.

4. Iterate over the stroke collection. For each stroke:
    a. Call the `Intersect` method on the `RegionCollection` object.
    b. The Intersect method has two out parameters: an array of `long` named `shapeIds` and an array of `int` named `intersectionTypes`.

       Each `long` integer in the shapeIds array is the `RegionId.longId` value for a region that the current stroke intersects in some way.

       The corresponding `int` of the `intersectionTypes` array indicates how they intersection. The available constants are defined in the `Intersect` class.

The following code snippet illustrates the way you can go about determining whether strokes on a page intersect any static regions defined on the page template:

```
foreach (KeyValuePair<long, Stroke> kvp in strokeCollection) {
    // Setting up for intersecting stroke with the RegionCollection
    const int MAX_INTERSECTS = 10;
    long[] shapeIds = new long[MAX_INTERSECTS];
    int[] intersectionTypes = new int[MAX_INTERSECTS];
    Shape s = (Shape)kvp.Value;

    int numberIntersects = regionCollection.Intersect(s, out shapeIds,
       out intersectionTypes, MAX_INTERSECTS);

    if (numberIntersects == 0) {
        Console.WriteLine("\n" + s + " did not intersect any shapes");
    }
    else {
         for (int i = 0; i < numberIntersects; i++) {
            string intersectString = "unknown";

            // For each intersection check what type of intersection
            switch (intersectionTypes[i]) {
                case Intersect.IntExternal:
                    intersectString = "DOES NOT INTERSECT";
                    break;
                case Intersect.IntClip:
                     intersectString = "CLIPS";
                     break;
                case Intersect.IntInternal1:
                     intersectString = "IS INSIDE";
```

```
                        break;
                   case Intersect.IntInternal2:
                        intersectString = "IS ENCLOSING";
                        break;
                   default:
                        intersectString = "THIS SHOULD NOT HAPPEN";
                        break;
                     } //end switch case

              RegionId regionId = new RegionId(shapeIds[i]);
              // Here we can pass either the long shapeIds[i] or
              // RegionId regionId object(works because of implicit cast)
              Shape intersectedShape = regionCollection.GetReferenceToShape(regionId);
              Console.WriteLine(s + " -=" + intersectString + "=- Id = " +
                  regionId.Id + " Shape = " + intersectedShape);
            }// end for
       }// end else
   }// end foreach
```

For a working desktop application that incorporates this code snippet, see `IntersectionExample.cs` in the `Samples` directory Livescribe Desktop SDK.

Miscellaneous Utility Classes

You should be aware of certain utility classes in the AFP API that provide various values that you need in your desktop application. A few of these utilities are:

Metrics

The Metrics class performs conversions between Anoto Units (AU) and universal standards of measurement, such as millimeters and inches. It also contains constants for standard page sizes from around the world, expressed in AU.

PenId

The `PenId` class has various conversion methods. For instance, it converts the 64-bit Pen ID to a string that represents the unsigned integer. Some classes provide a `PenSerial` property for you as a convenience. In other contexts, you will have to use the `PenId` class to convert the Pen ID to a string.

Note: You can pass the PenId object to a function that takes a `ulong penid`. This can be done because of an implicit cast of `PenId` to `ulong`. For example:

```
PenId pen = new PenId("XXX");
PageAddress pa = new PageAddress("2208.1.2.2");
lsDocument.SaveStrokes(pa, strokeCollection, penid);
```

Page Address

A page address is the unique address that identifies the dot pattern used for the page instance. The `Page Address` class allows you to manipulate page addresses.

Convenience methods allow you to convert from the 64-bit integer that represents a page address to a dot-separated address more easily understood by human beings. A dot-separated address looks like this: 12.10.7.8. That address uniquely identifies the following page in the Anoto dot pattern:
Segment 12, Shelf 10, Book 7, Page 8

Address Calculations

The PageAddress class has methods for performing addition and subtraction of page address. These can be useful when determining the number of pages between two addresses that are in different books or shelves. For instance: the `PageAddress.Subtract` method can tell you how many pattern pages exist between 12.10.7.12 and 12.10.8.1.

You can also perform page arithmetic with overloaded + and − operators, as in the following code snippet:

```
PageAddress p = new PageAddress("2280.2.1.2");
PageAddress p2 = p + 12;
```

Validating an AFD

It is a good idea to verify that an AFD you are using is valid before writing a desktop application to process it. The quickest way to determine validity at edit time is the following:

1. Open the AFD in the Livescribe Paper Editor.
2. Verify that the properties are set as you expect.
3. Verify that all images display properly.
4. Verify that regions have the Region IDs that you expect.

5. Examine the Application Mapping to see that the Application IDs are what you expect.

Creating a Page Viewer

You can create a desktop application that views the pages of an AFD, , using the Pen Data API and the AFP API. For a basic working page viewer, see `StrokeRendererForm.cs` in the Sample directory of the Livescribe Desktop SDK.

You must create bitmap versions of all graphic images that you will be using. Use of Image Magick and Ghostscript is a good public-license solution to generating bitmaps from your EPS files. Many proprietary image editors exist that create, edit, layout, and convert EPS files. Adobe Illustrator is an excellent example of such a tool. Be sure to choose a tool that supports alpha-blending.

The key to creating a page viewer application is the `PenData.Drawing.Renderer` class. Follow these general steps:

1. Create a `Document` object, passing the path of a ZIP file that contains penlet data or AFD data.
2. Create a `Renderer` object, passing the `Document` object to the constructor.
3. Implement your own `onPaint` method, which will be called by the system.
   a. The `PaintEventArgs` parameter passed to you by the system will contain a `Graphics` object. Call the `Save` method on that object. It will return a `GraphicsState` object.
   b. You can call the `ScaleTransform` method on the `Graphics` object. Call other methods on the `Graphics` object, as needed.
   c. Call the `Render` method on the `Renderer` object, passing the `Graphics` object. This renders the current graphics context to the screen.
   d. Call the `Restore` method on Graphics object, passing the saved `GraphicsState` object.
4. In the event handlers for your page viewer's UI controls, set the following properties of the Renderer object as needed. Their effect will become visible when the `Render` method is next called.

Property	**Description**
PageNumber	Integer representing page number of the page instance currently being displayed by the `Renderer` object. The CopyNumber property is 0, by default. To display another page, set the value to another page number.
DrawPaper	Boolean property that controls the background color of the page template.
DrawImages	Boolean property that controls whether background iimages from the page template are rendered.
DrawStrokes	Boolean property that controls whether any strokes written on the page instance are rendered.
DrawControls	Boolean property that controls whether the images of the various paper controls are rendered. You will often choose **not** to render them, since the penlet that provides the functionality of the paper controls is not running on your desktop computer.

Tip 1: Bitmap images must have an alpha channel to take advantage of alpha blending, provided by the AFP API.

Tip 2: Remember that the z-order of a `GraphicsCollection` runs in the reverse direction of the z-order of a `RegionCollection`. See Direction of the GraphicsCollection Z-Order.