2010

Gianni Van Hoecke
Maarten Meuris
Patrick Lauwaerts

University College Ghent
Faculty of Applied Business Mercator

LMS DESKTOP ASSISTANT

Manual for developers

Contents

1 Foreword

This manual is for developers. If you want help about using the program, please read the other manual.

You want to implement your own e-learning platform into the program? Or you want to implement your own document type? This manual will guide you through the necessary steps to make it work.

There are some requirements to start developing:

- We expect that you already have some knowledge about Object-Oriented programming. Knowledge of the Microsoft C# programming language is a plus.
- You'll need Microsoft Visual Studio 2010 and the Microsoft .NET framework 3.5 (http://www.microsoft.com/visualstudio/).
- The source code of *LMS Desktop Assistant*. Download at http://sourceforge.net/projects/lmsda/files/

Thank you for your interest in this project.

Gianni Van Hoecke

Maarten Meuris

Patrick Lauwaerts

2 Program structure

If you want to start developing for this project, you'll need to understand the structure of the source code first. We'll give a brief walkthrough.

2.1 Three layer model

LMS Desktop Assistant uses the three layer model. So, if we separate each layer by its functionality, then we'll see the following structure:

- **The user interface layer:**
  The user interface layer is the layer in which users interact with the application.
- **The domain layer:**
  The domain layer is the layer that the user interface layer and the persistence layer use to communicate with each other.
- **The persistence layer:**
  The persistence layer is basically a gate to the outside. It exports and imports the data *LMS Desktop Assistant* needs.

2.2 Patterns

LMS Desktop Assistant uses patterns. There are two important ones.

The first one is the Singleton pattern in the Domain Controller. It basically restricts the instantiation of the Domain Controller to one object.

The second one is the model view controller. This pattern isolates the logics from the user interface, enabling the possibility of multiple user interfaces.

2.3 Multithreading

Any application with actions that can take a while to complete uses multithreading. Hence, *LMS Desktop Assistant* uses multithreading.

If a user clicks on a button, a method in the controller will be called. Because some methods can take a long time to complete, the user interface will become unresponsive until the operation is finished.

To resolve this issue we've implemented multithreading, so the user will get the necessary feedback on what the program is doing.

Let us take an example. Open up the login form and look at the code behind the "Ok" button:

```
[…]
this.progressBar.Visible = true;
this.progressBar.MarqueeAnimationSpeed = 100;
this.setComponentsEnabled(false);
String[] para = new String[]
                    {txtUsername.Text, txtPassword.Text};
Thread thread = new Thread(
                    new ParameterizedThreadStart(
                       doLogin));
thread.SetApartmentState(ApartmentState.STA);
thread.Start(para);
[…]
```

First, we set the progress bar visible and start its animation. Then we set all the controls on the form disabled.

We're using a thread that's using parameters, so we need to declare an array with the values we want to pass on. Then we instantiate the new parameterized thread. "doLogin" is a private method we'll explain later. Before we start the thread, we set its apartment state to STA; it defines the thread as a single-threaded apartment.

If we take a look at the "doLogin" private method:

```
private void doLogin(object o)
{
   this.domainController.fireState(State.BUSY);
   String[] parameters = (String[])o;

   if (this.domainController.validateUser(
         parameters[0], parameters[1]))
```

```
        this.Invoke(new invoke_delegate(this.Close));
    else
        this.Invoke(new invoke_delegate(showLoginError));

    this.domainController.fireState(State.READY);
}
```

The array we've declared a while back is passed as an object, so we'll need to cast it back.

The following line will notify all user interfaces (MVC) that an action is taking place, and it needs to update according to the state.

```
this.domainController.fireState(State.BUSY);
```

Then we'll call the "validateUser" in the controller. This action takes a while to complete because it will communicate with the e-learning platform.

If that action is completed with success we invoke the method to close the form. You must use the "invoke" method because you're working in a different threat. You can't access GUI elements in a different thread because it's not safe thus an exception will be thrown.

The form has closed, but the thread is still running so we can still execute the last line of code:

```
this.domainController.fireState(State.READY);
```

The UI will be notified that the action is completed and the user will be logged in.

3 Platform

LMS Desktop Assistant is so designed that each platform can be added with barely any modifications to the existing source code.

We'll explain how to add a new platform with an example. In our example, we'll implement the platform *MyPlatform*, which is only accessible by POST requests.

3.1 Platform package and main platform class

Inside the "persistence/platform" package, we'll create a new package (folder), with the name of the new platform. In our example this would be "myplatform".

In the newly created package, we'll need to add a new C# source file. Name it according to the platform. "MyPlaform.cs" in our example.

Our newly created "MyPlaform.cs" class needs a constructor with a *Login* object as parameter. This *Login* object contains the user information. We also specify the suffix of the URL the user gets after a successful login.

```
class MyPlatform
{
   private const String PORTAL = "/portal.php";
   private Login login;

   public MyPlatform(Login login)
   {
      this.login = login;
   }
}
```

Now, to identify the class as being a platform, we'll need to make it inherit the abstract class "TargetPlatform". Do this by adding the following: (in bold)

```
class MyPlatform : TargetPlatform
```

And implement it:

Figure 1: we're implementing the abstract class.

3.2 Platform declaration

Now, to add the newly created class to the program, we'll need to create a new class inheriting "TargetPlatformInfo". This class will identify the platform, and allow the program to access its constructor. We'll put it inside the "persistence/platform/myplatform" package, and call it "MyPlatform_PlatformInfo.cs".

This class will contain 3 main elements:

- The name of the platform
- The encoding normally used on the platform
- The factory function linking to the constructor of the class

Besides those, there will be some other settings concerning the platform behaviour.

Again, we make it inherit the abstract class:

```
class MyPlatform_PlatformInfo : TargetPlatformInfo
```

If you let Visual Studio implement the abstract class, You'll immediately see the data you need to fill in. First is the platform name:

```
public override String getPlatformName()
{
    return "MyPlatform";
}
```

Next, we got the text encoding on the platform website:

```
public override String getPlatformEncoding()
{
    return "utf-8";
}
```

The “factory” function, which links to the correct constructor, is really simple in our case:

```
public override TargetPlatform factory
                        (Service service, Login login)
{
    return new MyPlatform(login);
}
```

For platforms with multiple possibilities for the Service, this function will contain a switch-case structure to choose a different class for each service.

Now these are done, make sure all other platform settings return a valid value. This class needs to be completely finished before we can continue.

And now the final step: adding our platform to the list of supported platforms.

Open the “TargetPlatforms.cs” file in the “persistence/platform” package, go to the “getSupportedPlatforms()” function, and add a new object of MyPlatform_PlatformInfo to the end of the array:

```
Private static TargetPlatformInfo[]
                            getSupportedPlatforms()
{
    return new TargetPlatformInfo[]
    {
        new Dokeos_1_8_5_hg_PlatformInfo(),
        new Chamilo_2_0_PlatformInfo(),
        new MyPlatform_PlatformInfo()
    };
}
```

This array is used for all actions concerning platform choices. If we compile and start the program, our new platform will immediately show up in the list. However, we’re still far away from any functionality.

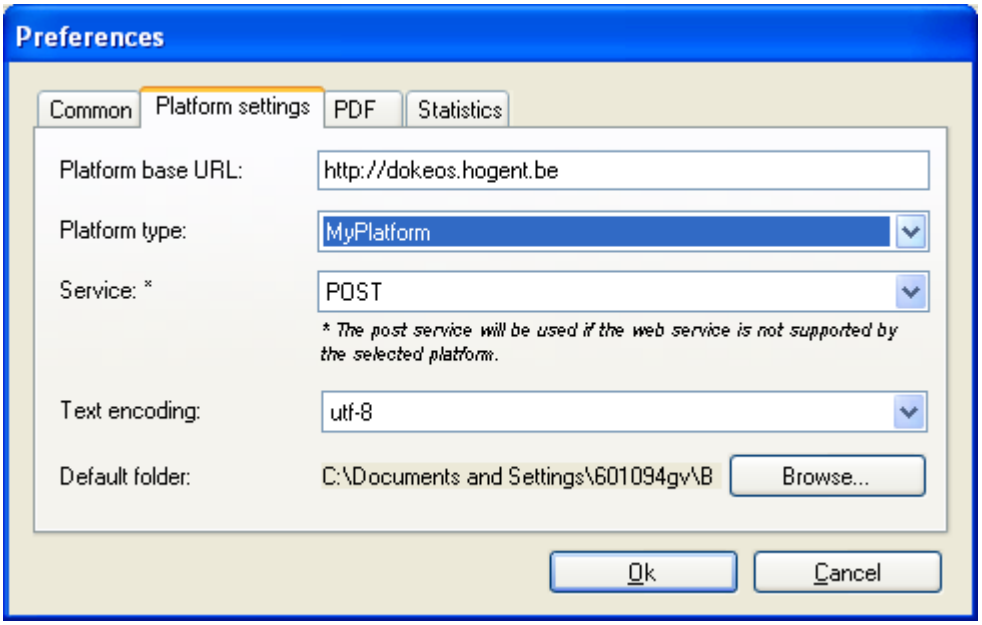

Figure 2: our platform will show up in the list.

3.3 Implementing the new platform

Now, let's go back to the our constructor in our "MyPlatform" class.

At the moment, the only thing it has is the Login object. Another important bit of information we need in order to communicate with the platform is the text encoding used on the platform website. Since we already have that in our platform information class, we can get it from there. We'll keep a local copy of the MyPlatform_PlatformInfo object, to have easy access to the other platform settings inside it.

After implementing the text encoding handling, our class will look like this:

```
class MyPlatform
{
   private const String PORTAL = "/portal.php";
   private TargetPlatformInfo platformInfo
                        = new MyPlatform_PlatformInfo();
   private Encoding encoding;
   private Login login;

   public MyPlatform(Login login)
   {
      this.login = login;
      String enc = DomainController.Instance().
         getSettings().getEncoding();
      this.encoding = Encoding.GetEncoding(
         (enc == null || enc.Equals(String.Empty)) ?
          platformInfo.getPlatformEncoding() : enc);
   }
}
```

Note that since the encoding is a user setting, we'll only use the default encoding from the MyPlatform_PlatformInfo object if there is no encoding configured in the settings.

Now you've got all the necessary methods to make the program work with your platform. Take a look at the existing platforms; you can use them as a reference. For this example, we'll just implement the login method. The rest is up to you!

We've got:

```
public override bool tryLogin()
{
   throw new NotImplementedException();
}
```

If we open the *MyPlatform* website in our internet browser, we'll see it asks us for a user name and password. If we take a look at the form details in the HTML source of the login page, we'll see that the user name form element is named "user_login", the password form element is named "user_password" and the submit button is named "submitAuth" with value "Login".

To emulate the login action in our class, we'll need to implement the following:

```
public override bool tryLogin()
{
   HttpSession httpSession =
         new HttpSession(this.encoding);
   httpSession.setRequestUrl
                  (this.login.getPlatformUrl());
   httpSession.addNameValuePair("user_login",
                  login.getUsername());
   httpSession.addNameValuePair("user_password",
                  login.getPassword());
   httpSession.addNameValuePair("submitAuth",
                  "Login");
   httpSession.sendPOSTrequestFromForm();
   String url = httpSession.getResponseUrl();
   if (url.Equals(
            this.login.getPlatformUrl() + PORTAL))
      return true;
   else
      return false;
}
```

Note that this example counts on a redirect after the login. If you are redirected to the same page after the login, you'll have to look for identifying elements in the returned HTML page that can give some kind of confirmation that the login succeeded. You can do that by scanning the page returned by "httpSession.getResponseFromServer()".

4 Documents

The way to add documents is virtually the same as the way platforms are added. Like Platforms, they works with an additional Info class for each document type.

We'll explain how to implement your own document type with an example. In our example we'll give you a start to implement support for OpenOffice Writer documents.

4.1 Document package and main document class

Inside the "persistence/document" package, we'll create a new package (folder), with the name of the new document type. In our example this would be "openoffice".

In the newly created package, we'll need to add a new C# source file. Name it according to the document type. "OpenOfficeDocument.cs" in our example.

This newly created class needs a constructor with two parameters:

- A String object that contains the full path to a document
- A Boolean object that indicates whether to remove all text styling when processing the document

```
class OpenOfficeDocument
{
   private String document;
   private Boolean plainText;

   public OpenOfficeDocument
      (String document, Boolean plainText)
   {
      this.plainText = plainText;
      this.document = document;
   }
}
```

To identify the class as a document for the program, we'll need to make the document class inherit the abstract class "SupportedDocument". Do this by adding the following: (in bold)

```
class OpenOfficeDocument : SupportedDocument
```

And implement it:

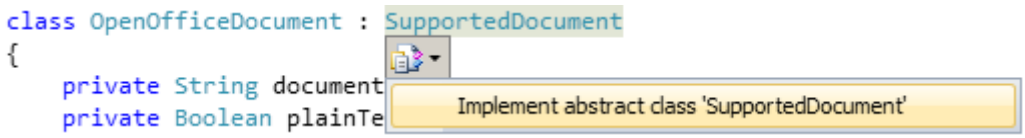

Figure 3: we're implementing the abstract class.

4.2 Document type declaration

Like with the platforms, we'll need to make an information class for our document type, and then add it to the list of supported document types.

The abstract class to inherit from for making the document information class is "SupportedDocumentInfo".

Again, this class contains a Factory to create objects of this type, and a number of other functions. The elements in this class are:

- A function to get all documents of this type that are currently opened
- An array of all supported extensions for documents of this type
- The factory function linking to the constructor of the class

So, we make an OpenOfficeDocumentInfo.cs class in the "persistence/document/openoffice" package, and let it implement "SupportedDocumentInfo". The resulting functions are these:

```
public override List<String> getAllActiveDocuments()
{
    throw new NotSupportedException();
}
```

At this moment, we won't implement a method to request a list of all currently opened OpenOffice Write documents; that would require communications with the OpenOffice program through an API like Uno. If

you want to implement that, be our guest. For this example, we'll just indicate that this document type doesn't support this functionality, by replacing the *NotImplementedException()* by a *NotSupportedException()*.

Next, we have this one:

```
public override List<String> getSupportedExtenstions()
{
    return new List<String>(new String[]{"odt"});
}
```

Unlike Word, which has 3 extensions, OpenOffice Write documents one only have one. So we add "odt" as only item in the array.

And finally, the factory:

```
public override SupportedDocument factory
                    (String document, Boolean plainText)
{
    return new OpenOfficeDocument(document, plainText);
}
```

As with the platforms, this returns an object of the class, created with the given parameters. Documents have no special cases in here though; this function will always contain just the constructor.

Now we defined our platform, we need to add it to the list of supported document types. That list can be found in the "SupportedDocuments.cs" class, located in "persistence/document".

```
public static SupportedDocumentInfo[]
                                getSupportedDocumentTypes()
{
    return new SupportedDocumentInfo[]
    {
        new MicrosoftWordDocumentInfo(),
        new OpenOfficeDocumentInfo()
    };
}
```

If we compile and start the program, our new document type's extension will show up in the list if we browse to a document file. However, we're still far away from any functionality.

4.3 Implementing the new document type

Now, let's go back to our newly created "OpenOfficeDocument.cs" class. At the moment, it has nothing but unimplemented classes. Now comes the real work; implementing all of them, to make the program work with your document type. Take a look at the existing document type; you can use it as a reference.

5 Adding a new language.

The procedure for adding a new language to LMS Desktop Assistant is quite simple. It doesn't require any changes in the program at all. In fact, language packs can be created without ever looking at the code.

For adding the example language "MyLanguage", the steps to follow are:

1) Go to the "language" subdirectory of the directory where the program is installed.
2) Make a copy of the file "English.ini", and rename it to "MyLanguage.ini".
3) Open the "MyLanguage.ini" file, and translate the text after the "=" character on all lines.
4) Make a copy of the file "English_blank.dotx", and rename it to "MyLanguage_blank.dotx"
5) Open "MyLanguage_blank.dotx" in Word. Make sure to open the actual file; the default behaviour of template files is to open as new document.
6) Adapt the style names in the document to the ones you've specified in "MyLanguage.ini".
7) Make a copy of the file "MyLanguage_blank.dotx", and rename it to "MyLanguage_full.dotx"
8) Add examples in "MyLanguage_full.dotx", following the ones in the original examples template.