C#でアプリを作ろう (前)

可不可

2010 年 11 月 5 日

mixi に「初心者の為の C#」というコミュニティがあります [4]。そこで公開したものにソースを付けたものです。

アプリを作るときに、新機軸を使ったりすると、途中で挫折したりします。また、新しいことをやるときに、いろいろ試行錯誤しますが、ようやく動くようになっても、その試行錯誤を忘れてしまうんですよ。「あれ？どうやったっけ」と悔やむこともたびたびです。きちんと記録すればいいんですけどね、つい忘れてしまう。

個人ブログも持っていませんので、「初心者の為の C#」で進行状況を記録することにしました。

1 最初の構想

1.1 何をするアプリか？

音楽ファイル (MP3) がたまってきました。それらにはタグを設定してあります。これらに以下の処理ができるアプリを作ろうとしています。

- 全音楽ファイルのタグを一括して印刷したい。
- ただ印刷するのも大変なので、Excel に出力して検索、並べ替えなどは Excel に任せる。
- タグに歌詞、画像もあるので、それらは 1 ページに 1 曲印刷して「歌集タイプ」にしたい。
- iTunes の中のファイルのタグも取得できるようにしたい。

実際に何から手を付けるか、問題を分けてみます。

- ファイルの追加はそうたびたびないので、起動のたびにタグを取得するのは時間のムダ。データベースを作っておいて、普段はそちらを使い、ファイル追加の時にデータベースを更新する。
- データベースといっても、単にタグを収納しておくだけなので、XML ファイルで充分だろう。
- Excel 出力はできれば印刷までも C#でコントロールしたいので、CSV ではなく、直接 XLS に書き出したい。
- iTunes 中のファイルは m4a という拡張子で、タグの形式も MP3 と違う。
- 歌集タイプの印刷は先送り。

タグを読む⇒データベース (XML) を作る⇒ Excel に書き出す
という順番に作って行くことになります。処理が独立しているのがいいですね。

個人で作るので、どんな方法を採ってもいいのですが、アジャイル開発で行こうと思います。これは、簡単に言うとつぎのようになります。

- 大体の目標を立てて、とにかく作り始める
- コードを書いたら必ずテストする
- 機能が不十分でも動くものを短期間でリリースする
- 動いているコードを洗練して行き（リファクタリング)、ドキュメントはコードとテストコードで代

用できるくらいにする

そこまでで一段落して、機能の再検討をして新しい目標を設定し、機能を付け加える後は、
機能追加⇒テスト⇒リファクタリング⇒リリース
の循環になります
開発環境

- VS 2010 アカデミックパック（プロと同じ）Expres 版でも使える（ハズ）
- Windows Vista
- NUnit 2.5　（テストツール）
- Visual NUnit（なくても OK)
- Quick Time （m4a のタグ取得用）
- MP3Lib （MP3 タグのライブラリ）などなど

2 タグを読む

まず、タグを読まなければなりません。MP3 なら、以前作ったクラスライブラリがあるので、それを使います。問題は iTunes にある m4a ファイルのタグです。

方法はいくつかあります。

1. iTunes Music Library.xml を調べる
   iTunes をデフォルトでインストールすると、ユーザフォルダの「ミュージック」というフォルダの中に iTunes Music Library.xml と iTunes Mususic フォルダができます。iTunes Music Library.xml には登録されているファイルのタグを含むデータが書いてあり、iTunes Mususic フォルダには実際のファイルが格納されています。この、iTunes Music Library.xml を解読してタグのみを取り出す方法があります。しかしこの XML ファイルは構造が独特で、一筋縄では行きません。
2. m4a ファイルのタグの規格を調べて読みだす。
   MP3 の時はこのテを使いました。規格が公開されていましたのでできました。しかし、m4a に関しては皆目わからない。ということであっさり却下。
3. iTunes 用のライブラリを使う。
   近くの大学図書館に「日経ソフトウェア」があるので調べたところ、「iTunes の COM インターフェイスを通してパソコンに接続した iPod を操作する」という記事を見つけました ([1])。Apple のサイトからマニュアルもダウンロードできます。でも、機能が多すぎ、マニュアルが英語なのでタグ取得がどこにあるかも分からない。サンプルを実行すると、iTunes が立ち上がってしまう。iTunes を使っている人は分かるでしょうが、立ち上がりに時間がかかるし、非表示にできないらしい。ということで断念。この日経の記事の中に.NET Framework で作った SharePod というフリーのライブラリが紹介されています ([2])。これはかなり有望だと思われますが、これも機能が多すぎ、マニュアルが英語ということで、これまた断念。
4. QuickTime を使う
   最後の頼みの綱で、CodeProject を調べたところ、「Read M4A tags in C#」というそのものズバリの記事を見つけました ([3])。（最初からここに来ればよかった）ソースも短く、英語でも何とかなりそうです。しかし、QuickTime のコンポーネントを使うのでフォームを使うことを強制されます。つまり、ふつうのクラスライブラリの DLL は生成できません。ということはテストツールが使えません。しかし、選択肢はこれしかないようなので、これを使うことに、不満足ながら決定！

これで最初のハードルのタグ取得の見通しがついたので、安心して最初の、対象ファイルの収集のコードを書けるようになりました。

3 ファイル名の収集

タグ取得クラスのイメージは、ファイル名を入れると各フレーム名（タイトル名、アーティスト名など）とその値のペアを Dictionary⟨string, string⟩ で出力する、という方針です。

その前に、タグ取得するファイルをどこから集めるか、が問題になります。これは、ユーザがフォルダを指定するということにします。

次に、そのフォルダの中からファイル名を抽出する過程が必要になります。普通のフォルダは問題ありません。iTunes は特定のフォルダにあるので、そのフォルダ以下のサブフォルダまで（再帰的に）取得すればいいようです。iTunes フォルダーはデフォルトでは各ユーザドキュメントのマイミュージックの”iTunes¥iTunes Musuc” 以下にあります。別フォルダにも作れますが、その下に作られる”iTunes¥iTunes Musuc” という構造は変わらないので、そこを攻めればいいようです。

次に、（タグ取得のための）指定フォルダ内のファイル名取得用のクラスのイメージは以下のようになり

ます。

1. フォルダを指定する。
2. 再帰的に取得するか指定 (iTunes は必ず再帰)
3. そのフォルダを走査して、m4a,MP3 プルパス名を取得する
4. 入力データ形式
   ファイル名と再帰か否か、のペアなので、Dictionary⟨string, bool⟩ にします。クラス名は、ファイルを集めてくるので GatherSongFile とし、メソッド名は List⟨ ファイル名 ⟩ GatherInFolder(Dictionary⟨string, bool⟩) としましょう
5. 出力データ形式
   当然、List⟨ フォルダ名 ⟩ でしょう

これで、コードを書く準備ができました。次回からテスト駆動方式でコーディングします。

これ以降の処理は、これから作る GatherInFolder クラスの吐いた List⟨ フォルダ名 ⟩ をタグ取得クラスに食わせて吐いたデータを順次 XML ファイルに書き出すという流れになります。

4 テスト環境を整える

アジャイル開発では、「変化の容認」という原則があります。しかし、今まで動いていたものがちょっとコードを変えたら動かなくなることが、製作途中で間々あります。そのため、変化があるたびに今までの動作の確認をしながら進めます。そのために「単体テスト」という方法があり、ツールとして xUnit（x には言語のシルシが入る）を使います。単体テストについてはネットにたくさんの解説があるのでここでは述べません。

4.1 Nunit のインストール

.NET で使うものは NUnit というものです。サイト [5] からダウンしてインストールしてください。私は NUnit-2.5.7.10213.msi をインストールしました。インストールすると、デスクトップに NUnit というアイコンができて、これは GUI ツール（テストランナーと呼ぶ）で、それなりに便利です。しかし、VS 中から使える方がさらに便利で、そのようなツールもあります。「Visual Nunit 2010」[6] というものです。

4.2 Visual Nunit 2010 のインストール

NUnit をインストールしてから、VS の「ツール」メニューから「拡張機能マネージャー」を起動し、Visual Nunit 2010 を選びます。後は適当にやってくれます。

使い方はいたって簡単で、「表示」メニューの「その他のウインドウ」の中に「VBisual Nunit」というエントリができてますので、クリックすると表示されます。もし、そのプロジェクトにテストコードがあれば自動的に読み込んで、登録されます。（オリジナルの NUnitGUI ツールでは手動で登録しなければならないのでちょっと面倒です）一番上にある緑の矢印をクリックするとテストが始まります。（こりゃ便利だ）

4.3 テストされるクラスを作る

「テストされる」ですので、実際に使われる DLL、EXE です。それをテストするのが NUnit で動くテストコードです。まず、テストされる方を作ります。新規プロジェクトで、クラスライブラリを選び、プロジェクト名（名前空間名）を GatherFile とします。（名前空間名とクラス名を同じにしてもコンパイルはできますが、あとで困ることがあるかもしれませんので、違う名前にします）デフォルトでできているクラスファイル名 (Class1.cs) を GatherSongFile.cs に変えます。デフォルトコンストラクタと、メソッドを作ります。コンパイルできるような最低限のコードを書いておきます。（このあたりは普通に DLL のときに

やっていることと同じです）

こんな感じになるでしょう。

```
namespace GatherFile
{
public GatherSongFile() { }

public List<string> GatherInFolder(Dictionary<string, bool> folder)
{
    List<string> retList = new List<string>();

    return retList;
}
```

4.4　試しのテストコードを書く

NUnit を使うにはソリューションエクスプローラの「参照設定」で追加のダイアログを開き、.NET の中にある nunit.framework を追加して、テストコードファイルの頭で using NUnit.Framework; と書いておきます。テスト用のファイルはまとめておくと便利なので、私は「新しいフォルダ」でテスト用フォルダを作り、その中にまとめてあります。テスト用クラスは Test_GatherFile とします。テストと指定するためにクラス名の前に [TestFixture] と書きます。同様にテストメソッドの前に [Test] と書きます。メソッド名はテストする条件を加えるので名づけるのに苦労します。しかし、ある時ユニコードなので日本語も通ることに気づきました。やってみると非常に具合がよろしい。

こんな感じになるでしょう

```
[TestFixture]
public class Test_GatherFile
{
    [Test]
    public void 初めのテスト()
    {
    }
}
```

ここまでのコードをテストしてみたスクリーンショットを付けておきます。手前にあるのが Visual Nunit 2010 です。右向き緑矢印をクリックするとテストが始まり、各テストコードが成功したら緑チェックが入ります。失敗したら赤×が付きます。この場合はテストでは何もしていないのですが成功してます。

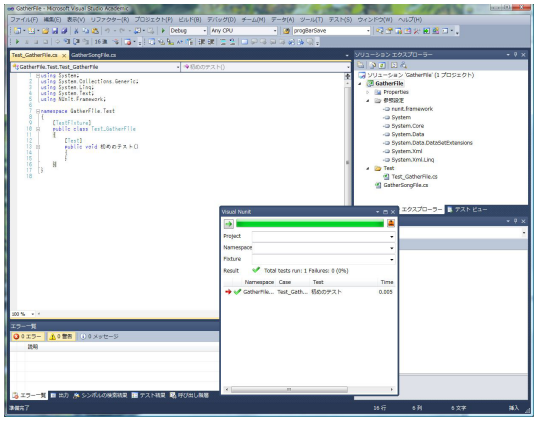

図 1　Visual Nunit でテストを実行する

5 テストコードを書く

5.1 GatherInFolder のテストコード

何もしないメソッドなので、空の引数を与えて GatherInFolder を呼んでも返ってくる値も空です。これをテストするには以下のコードです。

```
1 public void 初めのテスト()
2 {
3     Dictionary<string, bool> parame = new Dictionary<string, bool>();
4     List<string> resList = new List<string>();
5
6     GatherSongFile gather = new GatherSongFile();
7     resList = gather.GatherInFolder(parame);
8 }
```

さっそく右向き緑矢印をクリックします。緑ですね。空の GatherInFolder がちゃんと終了したということです。返り値が正しいかが分からないと正しく動作しているか分かりませんが、とにかくエラーも出ず最後まで実行はされているようです。

返り値を確かめるために Assert というコマンドがあります。この場合、返ってくる List¡string¿の string の個数はゼロです。これで確かめます。前のパラメータは期待値、後の方は実際の値になります。

次のようになります。

Assert.AreEqual(0, resList.Count);

これを 21:24 2010/11/05 加えて、テストします。もちろん緑ですね。

Assert.AreEqual(1, resList.Count);

とすると、当然赤です。正しいけど何か気になります。やはり緑にしたいものです。というわけで緑になるようなコードが自然に書けてしまう、という次第。

Assert のメソッドはいろいろありますが、当分は AreEqual で間に合います。慣れてくればケースに合わせたメソッドを選べるようになるでしょう。

環境も整ったところで、コードを書く前に、テストコードを書きます。これがテスト駆動開発 (TDD) と呼ばれるゆえんです。

テスト項目は

1. 指定フォルダが存在しない場合
2. 指定フォルダが空の場合
3. ルートフォルダのみから MP3、M4A ファイル名を取得
4. ルートフォルダ以下のフォルダから〃を再帰的に取得
5. iTunes Music フォルダの場合は必ず再帰

くらいでいいでしょう。

テスト環境でまだ残っているものがあります。テストデータのフォルダとファイルです。上のテストをするためには、適当なフォルダ（今回は F:¥Test）を作り MP3、M4a ファイルを入れます。それ以外にも別種類のファイルも入れます。再帰的に収集するので、その下にサブフォルダも作り、ファイルを入れます。もうひとつサブフォルダを作りますが、空にしておきます。

テストデータのフォルダー

- F:¥Test¥(各種ファイル)
- F:¥Test¥Non(空)
- F:¥Test¥Any(各種ファイル)

各種ファイルとは MP3、M4A ファイルと共にその他のファイルも入れておきます。

GatherSongFile の詳細テスト項目 1,2 の場合、List¡string¿.count=0 を返すだけにします。後半の処理に影響も出ませんので。

メソッド名

- Test フォルダなし
- Test フォルダ空
- Test ルート
- Test ルート再帰
- TestITunes

テストコード・ソースは次で公開します。このテストが全てグリーンになるようにソースを書いてゆきます。

6 テストコード・ソース

今回は詳しくソースを公開します。Assert.AreEqual の期待値はテストデータの状態で適宜変更してください。

```
1  private Dictionary<string, bool> inpTestData
2      = new Dictionary<string, bool>();
3
4  [Test]
5  public void Testフォルダなし()
6  {
7      List<string> resList = new List<string>();
8      inpTestData.Add(@"F:\Test\XYZ", true);
9
10     GatherSongFile gather = new GatherSongFile();
11     resList = gather.GatherInFolder(inpTestData);
12
13     Assert.AreEqual(0, resList.Count);
14 }
15
16 [Test]
17 public void Testフォルダ空()
18 {
19     List<string> resList = new List<string>();
20     inpTestData.Add(@"F:\Test\Non", true);
21
22     GatherSongFile gather = new GatherSongFile();
23     resList = gather.GatherInFolder(inpTestData);
24
25     Assert.AreEqual(0, resList.Count);
26 }
27
28 [Test]
29 public void Testルート()
30 {
31     List<string> resList = new List<string>();
32     inpTestData.Add(@"F:\Test", false);
33
34     GatherSongFile gather = new GatherSongFile();
35     resList = gather.GatherInFolder(inpTestData);
36
37     Assert.AreEqual(0, resList.Count);
38 }
39
40 [Test]
41 public void Testルート再帰()
```

```
42 {
43     List<string> resList = new List<string>();
44     inpTestData.Add(@"F:\Test", true);
45 
46     GatherSongFile gather = new GatherSongFile();
47     resList = gather.GatherInFolder(inpTestData);
48 
49     Assert.AreEqual(0, resList.Count);
50 }
51 
52 [Test]
53 public void TestITunes()
54 {
55     List<string> resList = new List<string>();
56     string iTunDir = Environment.GetFolderPath
57         (Environment.SpecialFolder.MyMusic) //各ユーザのミュージックフォルダ
58             + @"\iTunes\iTunes Music";
59     inpTestData.Add(iTunDir, false); //trueでも同じになるはず
60 
61     GatherSongFile gather = new GatherSongFile();
62     resList = gather.GatherInFolder(inpTestData);
63 
64     Assert.AreEqual(0, resList.Count);
65 }
```

後はテストがグリーンになるようにコードを書いてゆきます。この、「書くうちに赤が緑になって行く」って言うのはなかなかモチベーションが上がります。コードも見せながら説明しようともしましたが、面倒なのでいずれ Vector あたりで公開するということで次に進めます。

7　単体テスト考

単体テストを使ってみて気づいたことをまとめてみました。

7.1　VS にもテスト機能はあるけど？

Visual Studio のプロ以上には単体テスト機能があります。NUnit と同じ単体テストをする機能ですが、コマンド名が違っています。コマンド名を変更すれば、NUnit と同じコードが使えます（たぶん）。それに、内蔵テスト機能は MS 製品に特化しているので、Assert 関係が充実しています。さらに、スケルトンだけですが、テストコードを自動生成してくれます。これは、結構便利です。

ところが、テストを実行するたびに、トレース関係のファイルが、どんどん増える。これも、トレースのためか、実行まで時間がかかる。単体テストは、コードを書いたらすぐ実行して、すぐ結果が出ないと面倒くさくなり、使われなくなります。時間がかかるというのは致命的です。でも、このテスト機能は、チーム開発の一部という扱いらしく、ソースのバックアップ、共有と一緒に使わないと本来の機能が発揮できないようです。

個人的見解ですが、今回のように、一人で単体の Windows アプリの開発では付属のテストより NUnit が使いやすそうです。

7.2　テストコードはドキュメントも兼ねる

テストコードを見ればわかると思いますが、使う方法が読み取れます。内容を忘れてしまったクラスでも、引数。返り値などは必ず使いますので、テストがグリーンなら、これで使い方が分かります。ドキュメントというのは案外信用ならないもので、書いた後にバグをフィックスしても小さな変更なのでドキュメントに反映していないことがあります。また、ソース中に書くコメントも、書かなかったり、変更しないでしまうこともあります。本当に信用できるのはコードのみで、それが動作することを保証しているテストコードが唯一、最新・正確なドキュメントということができます。

7.3　デバッガは不用か？

NUnit を使えば、プログラムを起動して、結果が分かります。ではデバッガはいらないんじゃないか、と思えますが、目的が違います。テストで期待しない結果が出た場合、デバグにはデバッガを使わなければなりません。

NUnit では、結果しか分かりません。ということは、

- private メソッドをテストできない
- メソッド内部の動作が見えない
- フォームを含むものはテストできない

という限界があります。

内部の挙動を知るにはデバッガしかありません。DLL を作っている場合、同じソリューションにもう 1 つウィンドウフォームのプロジェクトを作り、そこから DLL を呼ぶようにします。そうすればデバッガを使うデバグができます。

テストとデバッガを比べれば、テストの方がやりやすいし、赤から緑にする、というのがモチベーションが上がります。そこで、考えを変えて、テストしやすいクラスにすればいいのです。テストコードから書き始めればどうでしょうか。

このあたりは私もまだはっきりしたことは分かりません。具体的にどうすればいいかというのも、まだ手探り状態です。

8 MP3 タグ取得クラスを作る準備

フォルダの中の MP3, M4a ファイルを集めて List⟨string⟩ に入れるところまでできました。

次は、この集めてきたファイルの１つ１つのタグを取得する番です。M4a はフォームを使うので先送りして、MP3 タグから始めます。

MP3 タグ取得はクラスライブラリを使うことにしています。Vector から「MP3 タグ・ライブラリ」[7] をダウンロードします。MP3TagLib.dll をしかるべきフォルダに入れて、参照を追加します。そして、名前空間を追加します。(using MP3TagLib;)

それから、テストも使いますので、nunit.framewwork も追加しておきます。（このあたりはもう、書かなくても分かるでしょう）

MP3TagLib に PDF のドキュメントが入っているので、目を通します。読出しはファイル名を引数にして Read で呼びだすだけです。

こんな感じです。

```
1  List<MP3Frame> tagData
2      = new List<MP3Frame>();
3  MP3Tag tag = new MP3Tag();
4  tagData = tag.Read(MP3ファイル名);
```

返り値がちょっと難物で、Dictionary⟨MP3Frame⟩ です。MP3Frame のメンバーで使うのは、frameID（タグ名)、dataObj（データ本体）の２つです。

frameID は FrameID の列挙型で、規格書のフレーム ID です。このままだと分かりずらいので、title, artist などの分かりやすい文字列に変える必要があります。一応、以下のように名づけました。

```
1  FrameID.TALB = "album";
2  FrameID.TPE1 = "artist";
3  FrameID.COMM = "comment";
4  FrameID.TCOM = "composer";
5  FrameID.TCON = "genre";
6  FrameID.USLT = "lyric";
7  FrameID.APIC = "pic";
8  FrameID.TIT2 = "title";
9  FrameID.TRCK = "track";
10 FrameID.TYER = "year";
```

FrameID を文字列に変換する、private string GetFrameName(MP3Frame frame) のような関数も必要になるでしょう。

dataObj は、写真データも含まれるので object 型です。今回は、写真データは必要なく、写真があるかないかの区別があればいいので、string にキャストします。”pic”の場合は”*P*”を返すことにします。

コメント、歌詞は複行の文字列なので、Excel に書き出すと巨大セルができるので、こちらも存在のフラグを立てるだけにします。”comment”は”*C*”、”lyric”は”*L*”とします。

というわけで、返り値は Dictionaryu⟨string(分かりやすいタグ名),string(文字列タグデータ)⟩ にします。

タグを設定していないファイルの場合、MP3 ではあるので、フルパス (”path”) と、フルパスは長くなりがちなので、ファイル名 (”fileName”) のみと、ファイルのタイムスタンプ (”createTime”) を文字列にして返すことにします。だから、タグがなくても、最低 3 個のデータが返り値の Dictionari の中にあることになります。

このクラスの骨組みを以下のように決めます。

```
1  namespace GetTag
2  public class GetTagMP3
3  メソッド
4  public Dictionary<string, string>GetTag(string fName)
5  （最低でも3個のデータを含む）
```

テスト項目は

- 普通のタグを設定したファイルを読ませて、タグの個数を確認
- MP3 以外のファイル（期待値 0）
- タグのない MP3 ファイル（期待値 3）
- 存在しないファイル（期待値 0）
- 返り値の中の Directry⟨T⟩ に”pic”があればそのデータは”*P*”か？
- 返り値の中の Directry⟨T⟩ に”comment”があればそのデータは”*C*”か？
- 返り値の中の Directry⟨T⟩ に”lyric”があればそのデータは”*L*”か？

これらがすべてグリーンならＯＫでしょう。

9　M4a タグ取得

先送りしてきた M4A タグの取得にかかります。QuickTime コンポーネントを使いますので、その準備をします。QuickTime（以下 QT）がなければ話になりませんので、インストールを確認します。デフォルトなら C:¥Program Files¥QuickTime にインストールされています。

フォームの準備 VisualStudio を起動して FileToXML というプロジェクトを作ります。参照設定で QT フォルダ中の DLL(4 コ) を追加します。次にコンポーネントを追加します。メニューバーの「ツール」下の「ツールボックスのアイテム選択」をクリックすると、アイテム選択のダイアログが出ますので、「COM コンポーネント」タグの中から QuickTimeObject にチェックを入れます。すると、ツールボックスに Apple Quicktime Control 2.0 というアイテムが出るようになります。この状態のスクリーンショットを添付しておきます。(図 2)

後は普通のウィンドウズアプリと同じで、Form に QT コントロールを D&D します。このコントロールは、画像再生のためか小さくならないので、Visible プロパティを False にして非表示にします。

テスト用のボタンを配置して、以下のコードを書きます。QT コントロールを渡して、タグデータを受けるクラスを呼んでいます。

```
1 private void btnRead_Click(object sender, EventArgs e)
2 {
3     m4aPath = @"F:\DEV2010\10M4A\smpl\smplM4A.m4a";
4     Dictionary<string, string> tagDic;
5     qtM4ALib m4a = new qtM4ALib(axQTControl1);
6     tagDic = m4a.GetTag(m4aPath);
7 }
```

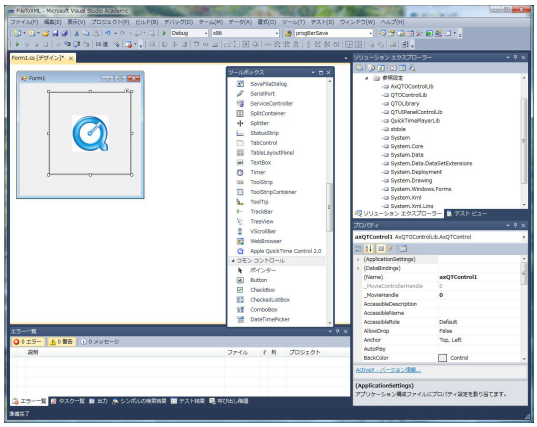

図 2　QuickTime コンポーネントを使う

M4A タグの取得クラスの説明は難しくないので、コードで代用します。

```
1  public class qtM4ALib
2  {
3      private AxQTOControlLib.AxQTControl QTControl;
4
5      public qtM4ALib(AxQTOControlLib.AxQTControl axQTControl) {
6            QTControl = axQTControl;
7      }
8
9      internal Dictionary<string, string> GetTag(string m4aPath)
10     {
11         Dictionary<string, string> Dic
12               = new Dictionary<string, string>();
13         QTControl.URL = m4aPath;
14         QTOLibrary.QTMovie mov = new QTOLibrary.QTMovie();
15         mov = QTControl.Movie;
16         if(mov == null)
17             return null;
18         //Get the file props
19         string fram = "";
20         fram = mov.get_Annotation
21             ((int)QTAnnotationsEnum.qtAnnotationArtist);
22         if (fram != null)
23             retDic.Add("artist", fram);
24         //同じように別のタグ取得
25         return retDic;
26     }
27
28 }
```

10　全体をまとめる

クラスもそろったので、全体を１つにまとめます。全体の流れは

10.1　ユーザの操作

1. タグを取得するフォルダを選ぶ (D&D)
2. そのフォルダを再帰的に処理するか (チェック付きリストボックス)
3. iTunes も含めるか (チェックボックス)
4. 出力する XML ファイルパス指定 (フォルダ選択ダイアログ&テキストボックス)
5. (アプリ起動)

10.2　アプリの挙動

1. 指定したフォルダからファイルを集める (GatherFile クラス)
2. 個々のファイルのタグを取得
   (GetTagMP3[名前空間：GetTag],GetTagM4a[名前空間：FileToXML])
3. XML ライター設定

4. 収集したファイルがある限り以下を繰り返す
   (foreach) ファイル毎
   ｜タグ取得 (GetTag クラス)
   ｜ (foreach) タグ毎
   ｜　 ｜ XML に書く
   ｜ (next) タグ
   (next) ファイル
5. ファイルがなくなったら XML ライター終了

10.3　全体アプリ・クラス詳細

名前：MakeTagXML
引数：まず、GatherFile を呼ぶので Dictionary⌊string, bool⌋
出力する XML ファイルパス (string xmlPath)
返り値：なくてもいいのですが、処理したファイル数にします。
DLL：QT **関係：**MP3TagLib.dll, GetTag.dll, GatherFile.dll を参照に追加

テスト項目
MP3 のみを xmlPath に書き出す（M4A は単体テストが使えないので）
Assert で個数を確認し、結果は XML ファイルを読んで確認します。

11　さいごに

後はコードを参照してもらうことにして、今回はこれで終わります。出力した XML ファイルはそのまま Excel に読ませることができますので、ユーザ（自分ですが）に見せることができます。それによってこれからの到達目標、これまでの機能追加などの変更点が出やすくなることでしょう。

残りの XML ファイルデータの検索、Excel 書き込みなどは、新たにトピを立てます。ソースの公開は、この文書を PDF 化して添付しますので少々お待ちください。公開したらここでお知らせします。

公開はしますが、まだ不完全で、リファクタリングが不十分です。もっとコードを洗練させるべきです。その際の動作確認にとってテストは充分な保障になるでしょう。

参考文献

[1] こだか かおる, 「iTunes の COM インターフェイスを通してパソコンに接続した iPod を操作する」, 日経ソフトウェア (2008.1 p97-105)

[2] SharePod , http://www.getsharepod.com

[3] jesseseger , Read M4A tags in C#, http://www.codeproject.com/KB//files/m4afiletags.aspx

[4] 初心者の為の C# http://mixi.jp/ view_community.pl?id=1428960

[5] NUnit Home , http://nunit.org

[6]「Visual Nunit 2010」 , http://visualstudiogallery.msdn.microsoft.com/en-us/c8164c71-0836-4471-80ce-633383031099

[7]「MP3 タグ・ライブラリ」http://www.vector.co.jp/soft/winnt/prog/se483898.html
http://ja.wikipedia.org/wiki/.NET_Framework