Department of Computing
Imperial College London

Final Year Individual Project

Bringing F# to the Masses

Functional Programming for Music Libraries

21st June 2011

Author:
Paul Clement

Supervisor:
Prof. Susan Eisenbach

Abstract

Music libraries are a great source of data, much of which the user can't do anything with. Although media players will show the user things such as play counts, there is no simple way to take this data and perform actions such as ranking your favourite artists or getting recommendations of artists you might like. Should you wish to find out more information about genres, artists or albums that are in your library, most media players expect you to do this manually, opening a web browser and searching the web to find what you want to know - these are both issues that this project aims to solve.

An application was created to generate statistics from the library data as well as producing rankings and recommendations for the user. As many people also listen to music on a mobile device, a Windows Phone version of the application was also created. An additional aim of the project was to evaluate the F# programming language - how difficult is it to use F# for this type of application, and how does it compare to other languages that are available.

The language proved to be very suitable for tasks of this type, not requiring any complicated workarounds to do things, and in some cases there were clear advantages to using the language over others. Users found that the application had features that they would use and generally liked the applications, although they were able to give suggestions of areas that could be improved and features that could be added.

Acknowledgements

Firstly, I want to thank my supervisor, Susan Eisenbach, for making sure that I was on track throughout the project and pointing me in the direction of various people and resources that were relevant.

I also want to thank my second marker, Michael Huth, for his encouraging words and helpful feedback from my outline report, and Don Syme, of Microsoft Research, for giving some helpful suggestions later in the project.

Finally, I want to thank my family and friends for their ongoing support and encouragement as well as those who participated in my user studies - I couldn't have evaluated my project without you!

Contents

Chapter 1

Introduction

This project has two main aims: to develop a piece of software that can provide additional information about music library contents and to evaluate the F# language, seeing how difficult it is to use to write real-world pieces of software.

1.1 Functional Programming for Music Libraries

Music libraries are a great source of data - song lengths, play counts, number of songs - and as such interesting information can be found through analysing them. Is a large proportion of your library all made up from the same artist? How much of your library do you *really* listen to?

It is also often the case that you want to know more about an album or artist. Is there a new album out? Are you missing any songs from an album? What artists are similar (and are they in your library)? What information is available about the artist themselves? There are numerous online services which can provide answers to these questions but why not have them all available in a single place, linked in with your music library to give you the information that you want?

The primary aim of this project is to provide an application that will interact with music libraries such as Apple iTunes and Microsoft Windows Media Player, and collect additional information from the Internet with which to enhance the listening experience. It will also provide analysis of music libraries, showing various statistics in a simple, easy to understand manner.

A functional language is suited to this project as it is largely to do with processing data. Very little state needs to be stored (apart from in the user interface), with data being pulled from various sources, both local and online. Some features of the language that need to be available are being able to interact with music libraries through the Component Object Model (COM) and being able to get data from the Internet. It also needs to be able to create and interact with a user interface, and hence a language that is functional but provides a large number of additional libraries is ideal.

The system was implemented on both Windows (on the .NET Framework) and on Windows Phone 7, to allow the user to get the benefits of the system whatever they use to listen to their music.

1.2 F# as a Language

This project also has a secondary aim: to evaluate F# as a language - how difficult is it to build a whole application, from the user interface to data processing?

At a first glance, F# looks as though it should be able to implement all parts of an application fairly easily, being a language that runs on the Microsoft .NET Framework with it's rich user interface libraries and easy interoperability with other applications through COM. By using F# for this project, I aim to show that it is not just a research language that can only be applied to a (relatively) small set of tasks and is as easy to use as other languages for producing fully functional real-world applications, pulling data from various sources, and displaying that information to

users in a clear and easy to consume manner. There is far more to the language than the idea of processing lists of tuples (although this is something that F# can do very well).

By implementing the system on Windows Phone 7 as well as on Windows, the ability of F# code to run across multiple platforms with minimal, if any, changes being required to the code will be demonstrated.

1.3 The Application

During this project, two applications were developed: one running on Windows under the Microsoft .NET Framework, and one running on Windows Phone 7. These applications largely have the same feature set, with a few minor differences specific to each platform.

Both applications have the following features:

- Provide statistics about the contents of music libraries, including total and average playcounts, durations and song counts.
- Allow the user to access additional information about the genres, artists and albums in their library, taking information from various online data sources.
- Provide a list of recommended artists through finding similar artists to the top artists in the library by various criteria.
- Provide rankings of top genres, artists and albums based on playcounts, song counts and total track durations.

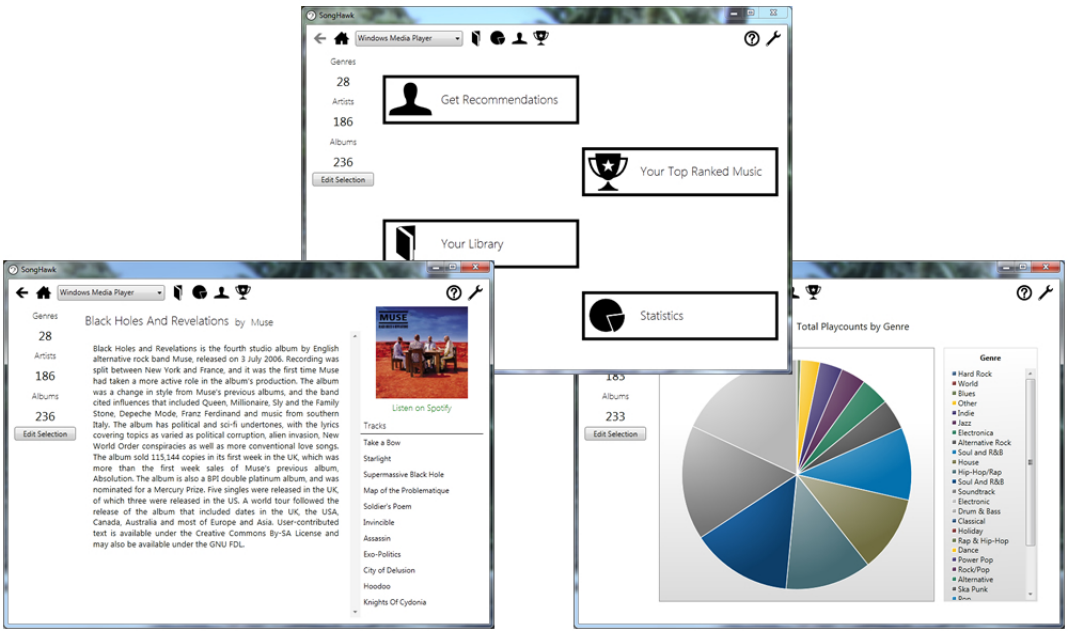

Figure 1.1: Screenshots of the Windows Application

The Windows Phone version of the application is limited to only supporting the default Windows Phone media library, and does not provide any configuration options to the user. The Windows version of the application has the following additional features:

- The user can choose between various library plugins to access libraries such as Windows Media Player or iTunes.

- On artist or album information pages, links are provided to allow tracks to be played from Spotify. This allows the user to listen to recommended artists that they do not already have or listen to albums that they do not own.

To allow the F# language to be evaluated, both versions of the application are written in F#, with a large amount of re-use of code between the two. Due to the platform differences, it was necessary to create separate user interfaces, each tailored to the specific platform, and to have separate library interaction code, although an abstraction layer was used to allow the code to work against any implemented library type.

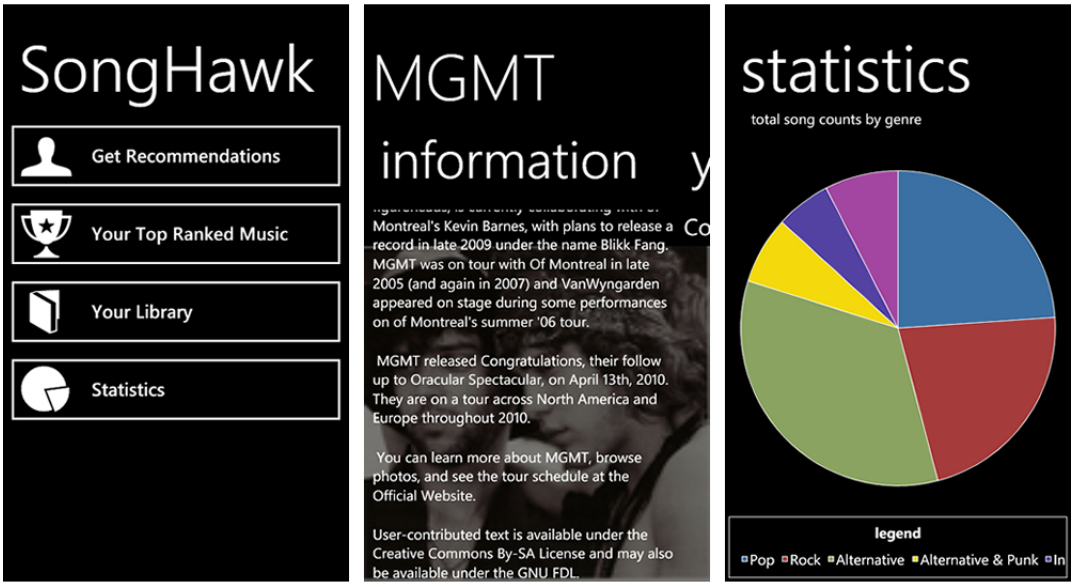

Figure 1.2: Screenshots of the Windows Phone Application

1.4 Report Structure

Throughout the rest of this report, we will discuss the various technologies and techniques that were required to complete this project.

The background information will cover various topics relating to the project, including language features and details of third party libraries and data sources that have been used.

We then cover the architecture of the overall system, followed by the design and implementation details of the various components of the system, both shared and platform specific. We also discuss the methods of testing that were used during the project.

Finally, we will evaluate the project, discussing whether it has been successful or not and determining the advantages (and disadvantages) of using F# to complete the project.

Chapter 2

Background Information

There are a number of different areas that I have investigated to get a full understanding of what has already been developed and how these relate to my project. They fall into six main areas: the F# language, application programming interfaces for various media players, methods available for accessing further information about the contents of a media library from the Internet, methods of visualising data, pluggable system architectures and running F# code on other platforms. My research into each of these different areas will be detailed in this chapter.

2.1 The F# Language

In this section, I will discuss a number of different features of the F# language that will be useful in the implementation of this project. The **multi-paradigm approach** of the language will help with interoperability with libraries and APIs, the **asynchronous programming support** will help with dealing with threads in a simple manner, particularly when used with UI technologies, **Active Patterns** will help when filtering and analysing library data, and **Type Providers** provide a simple way to access data from the Internet without having to deal with data formats and communication protocols. Each of these features will now be covered in more detail.

2.1.1 A Multi-Paradigm Language

As stated in *Foundations of F#*[29], "with F#, you can choose whichever paradigm works best to solve problems in the most effective way." Although F# is a functional language, it also allows you to program in an imperative or object-oriented style, so you can use the strengths of the different paradigms together.

This multi-paradigm approach also means that the language can benefit from the full features of the .NET Framework, including the multitude of libraries and the ability to interact with code written in other .NET languages such as C#. As a first class .NET language, "F# embraces .NET techniques such as dynamic loading, dynamic typing, and reflection, and it adds techniques such as expression quotation and active patterns."[9]

2.1.2 Asynchrony and Parallelism

F# has language support for asynchronous programming[11]. This allows work to be done in a background thread without having a large amount of code to handle creating threads, callbacks and dealing with exceptions.

Asynchronous blocks of code can be created using the `async` keyword, and these blocks of code can then be run using `Async.Start` to run it in a background thread, or `Async.RunSynchronously` to run it whilst blocking the current thread (most useful when used in conjunction with the

`Async.Parallel` construct). An example of an asynchronous block of code which sets the text of a label every five seconds and sleeps in-between can be seen in Program 1.

Program 1 Non-blocking code using F# asynchrony support

```
async {
    asyncLabel.Text <- "Async call starting!"
    do! Async.Sleep 5000
    asyncLabel.Text <- "At 5 seconds!"
    do! Async.Sleep 5000
    asyncLabel.Text <- "At 10 seconds and done!"
} |> Async.Start
```

F# has some variations on standard keywords (`let!`, `use!`, `do!`), which are similar to the `await` keyword in C# 4.0 and tell the thread to wait for the response from an asynchronous method without blocking. These allow asynchronous methods to be consumed by the program in the same way as any other piece of code, removing the need for callbacks and all the other pieces of code that are generally associated with asynchronous programming. This also means that exceptions that are thrown during execution of the asynchronous methods can also be handled using a `try ... with` block as any other exception would be handled in the language.

Language level support for asynchrony allows more expressivity than the standard asynchronous programming model as it is not necessary to deal with how work will be separated across multiple threads, allowing you to use programming styles that would otherwise be difficult to use such as sequencing, looping, recursion and pattern matching.

This also makes it easy to make use of patterns for concurrent and reactive programming in F# applications, for example:

- Parallel composition can be used to execute multiple threads concurrently, all working towards a single result as shown in Program 2. Although the method call for each of the four elements in the list comprehension takes five seconds, the whole parallel method call only takes five seconds, with very little additional code being necessary to provide this performance enhancement.

  The `Async.StartChild` method can also be used to produce parallel execution, although this leads to more verbose code.

- Reactive agents that use state machines can be used to perform actions, with messages being sent to the agent and queued, with messages being reacted to in an asynchronous manner with the caller waiting for a response.

- Reactive user interfaces can also be created whereby UI code waits for events to be fired, with all code being programmed as though it was in the UI thread but without the side effect of the UI blocking whilst waiting.

Program 2 Use of the method allowing multiple threads to be spawned and executed concurrently

```
let result =
    Async.Parallel [ for i in [1;2;3;4] -> return 15iAfter5seconds i ]
    |> Async.RunSynchronously
// result = [15;30;45;60]
```

2.1.3 Active Patterns

Pattern matching is an important part of functional programming, but in its standard form is unable to operate on abstract data types as the underlying representation is hidden.[10] Simple pattern matching on abstract types can easily turn into nested `if` statements as evaluation of the

object is required. This can be solved by exposing the representation of the object, however this is undesirable, particularly with object-oriented programming where abstraction is used to realise encapsulation.

F# has a lightweight language extension, Active Patterns, which aims to solve this problem by allowing special functions to be defined and used as part of the pattern. These can be defined to operate on any type and can be statically checked for completeness and redundancy of matches. Active patterns essentially create `choice` types which can then be pattern matched against, effectively allowing us to view objects as though they were defined in such a manner.

A variety of different types of pattern can be created using active patterns:

- Simple Total Patterns - the pattern is used to decompose the data into a number of sub-cases. (See Program 3)
- Partial Patterns - the data will be decomposed or the pattern will return `None` and matching will then occur on the next pattern. (See Program 4)
- Parametrised Patterns - the pattern can take multiple parameters, e.g. matching a string against a regular expression.

Only the first of these supports analysis of completeness and redundancy, as by their nature, partial and parametrised patterns are incomplete. Active patterns can also use other patterns in their definition, and it is possible to ensure that data matches multiple patterns through the use of "both" patterns e.g. `pattern & pattern`.

Program 3 A simple pattern for extracting the parts of a complex number.

```
type Complex = { RealPart : int; ImaginaryPart : int }

let (|Rect|) (x:Complex) =
    (x.RealPart, x.ImaginaryPart)

let add (Rect(ar,ai) : Complex) (Rect(br,bi) : Complex) =
    { RealPart=ar+br; ImaginaryPart=ai+bi }
```

Program 4 A partial pattern for checking if a string can be converted to a double.

```
let (|ADouble|_|) (s:string) =
    let i = ref 0.0
    if Double.TryParse(s,i) then Some !i
    else None

let tryToParse = function
    | ADouble x -> x
    | _ -> Double.NaN
```

2.1.4 Type Providers

With current programming languages there is an inherent problem that the world is information-rich and modern applications are information-rich, however programming languages are information-sparse so there is no specific way to handle this information. A solution to this problem is being developed for F# through the use of "Type Providers".[31].

Currently, to have a strongly-typed set of data in a program you might create a record type, then populate some records, and then add metadata to link between objects. This process is slow, with types having to be created and datasets being populated before you can make use of it. Type Providers are intended to provide all of the benefits of a strongly typed dataset which are supported like any other F# type without having to go through this process.

There is a large amount of organised data in the world from which an enormous number of types can be created. This information is very interlinked and often uses freely available data sources such as Wikipedia. To access this data, you need to request it and parse it, having a good understanding of the format that the data is represented in. Code generators are available for many of these types of data source, however these tend to generate a lot of code. Type providers allow this data to be accessed without having to understand the underlying data format or generate large amounts of code.

Type providers can be produced by the data provider from a schema of the data, and the created types can be statically checked, with the responsibility for soundness being placed on the data. As the types are lazily evaluated, there is no need to load them all when writing and compiling, so even if there are many thousands of types available, only those that you want to use will be available. It is also possible to use type providers with live datasets, and as it is an open architecture, type providers can potentially be used with any type of data source. There are already a number of data sources available to be used with this system, including datasets from the Windows Azure DataMarket (`https://datamarket.azure.com/`) and Freebase (`http://www.freebase.com/`).

2.2 Media Player APIs

To ensure that this project can be useful to many people, it was decided to allow interaction with various different media libraries, and the application was developed with this in mind, allowing new types of library to be added later and 'plugged in' to the system. As a starting point, the APIs for Apple's iTunes and Microsoft's Windows Media Player were investigated.

2.2.1 iTunes

Research into accessing iTunes library information from F# began by reading a blog post by Cameron Taggert[32]. This post demonstrated that this was possible and also demonstrated the use of some extension methods allowing additional information to be easily accessed through a COM API that had been generated from the iTunes executable, for example Program 5 shows how to find tracks for which the files that cannot be found.

Program 5 Sample code for finding missing files from the iTunes library in F#[32]

```
let tracksMissingFiles =
  app.LibraryPlaylist.Tracks.Files
  |> Seq.filter (fun t -> false = File.Exists t.Location)
  |> List.ofSeq
```

Further investigation into communicating with iTunes from .NET led to a blog post by Dan Crevier[8]. This revealed that Apple provide a COM object with each version of iTunes, which can simply be added as a reference from a .NET application to provide the ability that is required. This post also provided a helpful link to Apple's iTunes COM SDK[14], from which you can download documentation for the API and also some sample JavaScript code for interacting with iTunes.

2.2.2 Windows Media Player

Research into accessing the Windows Media Player library also revealed that there is a COM API available, with full documentation available from the Microsoft Developer Network.[27] This COM API provides similar behaviour to the iTunes API. Program 6 shows how information from the Windows Media Player library can be retrieved given the full path of a file (code taken from the MetadataBackup open source project[3]). Although this code snippet is written in C#, the same API calls can be used in F# due to the underlying .NET Framework.

Program 6 Finding library data about a file given a path in C#[3]

```
WMPPlaylist playlist = Player.mediaCollection.getByAttribute("SourceURL",
        fullFilePath);
if (playlist.count == 1)
{
    IWMPMedia media = playlist.get_Item(0);
    string mediaType = media.getItemInfo("MediaType");
    LogVerbose(String.Format("{0} found in media library - media type: {1}",
        fullFilePath, mediaType));
    if (mediaType == "audio") // TODO: allow other media types
    {
        BackupLibraryData(playlist.get_Item(0), writer); // xDoc);
    }
}
else
    LogVerbose(String.Format("{0} NOT found in media library ({1})",
        fullFilePath, playlist.count));
```

2.3 Sources of Data

2.3.1 Last.fm

Last.fm provides a web service which can be queried using either REST or XML-RPC, both of which return data encoded as an XML document. It provides methods for accessing various pieces of information, such as artist, album or track information and if a user has been authenticated with the service, it can also interact with Last.fm features such as writing in a shoutbox or adding a track play to the user's profile.[21]

As this project is more concerned with consuming data rather than creating data, this web service was only used to get additional information about the contents of the user's local music library, such as finding additional information about albums and artists.

2.3.2 Freebase

Freebase provides a large amount of data on various different topics that can be queried using the Metaweb Query API.[15] For this project, data is consumed from the Music section, which at the time of writing has over 30 million facts and over 9 million topics, with data available on topics including artists, albums, composers, genres and record labels.[16]

To read data, the `mqlread` service is used, with a JSON (JavaScript Object Notation) request which returns the requested data encoded as JSON.

Freebase is also currently developing an RDF service for accessing data and finding connections between pieces of information as a graph data structure.

2.3.3 MusicBrainz

MusicBrainz provides a database containing music metadata that is maintained by the MusicBrainz community. A web service is provided using a REST API to allow data to be accessed, and information about artists, releases and tracks are all available. Accessing data on an individual item requires the use of a MusicBrainz ID, which can be found by calling the service with filtering requests.[6]

A .NET library, MusicBrainz Sharp[5], is also available for accessing MusicBrainz data without having to create the XML requests and parse the responses. This provides a strongly typed representation of the data, with methods provided for accessing the various types of data that are

available using a MusicBrainz ID or by using strings for querying the database. An example of the use of this library can be seen in Program 7.

Program 7 Get the first 10 release names and types by The Beatles using the MusicBrainz Sharp library

```
let mb = Artist.Query("The Beatles").PerfectMatch()
printfn "%s - %s" (mb.GetName()) (mb.Id)

for a in (mb.GetReleases() |> Seq.take 10) do
    printfn "%s - %s" (a.GetTitle()) (a.GetReleaseType().ToString())
```

2.3.4 Spotify Metadata API

The Spotify Metadata API provides a REST web service[22] which returns limited data about items that are available on Spotify. Responses from the service can be sent as XML or JSON, and this can be specified in the web service URL or using HTTP request headers.

This could be used for comparing the songs in a library with those on Spotify to find any missing tracks, or to find other releases by the same artist. Another use of this API would be to find Spotify URLs to share with other people, for example through social networks.

2.3.5 Comparison of Data Sources

These data sources all provide a large amount of information, and some of this data can be found from multiple of these sources. Each of these sources do however provide specific information that is not available from the others.

All of these data sources can provide lists of releases by artists, as well as track listings and basic track information such as names and durations.

There is also overlap between some of the different sources, Freebase and Last.fm can provide a biography of an artist as well as listing genres for the artist. They can also provide background information for releases. MusicBrainz and Freebase can both provide background information about record labels.

Each of these sources also provides some information that cannot be found in any of the others:

- Last.fm can provide additional information about tracks, including a description and what albums it appears on. It can also provide a list of similar artists, genres for a release and images for artists and album artwork.
- MusicBrainz can provide the type (album, single, compilation etc.), language and worldwide release dates of an album as well as the type (distributor, producer etc.) and any relevant dates (e.g. founding) of a record label.
- Freebase can provide a description of a genre.
- The Spotify API can provide URLs for playing tracks, albums or artists in Spotify.

2.4 Data Visualisation

2.4.1 Windows Presentation Foundation

By default, there are two user interface frameworks available from F#, WinForms or Windows Presentation Foundation. WinForms allow you to create applications based on forms and controls[29], with the whole of the UI being generated procedurally. Windows Presentation Foundation (WPF) allows you to design the UI using an XML-based language called XAML (see Program 8).

For this project, it makes sense to use WPF as it removes the need to write as much F# code just for generating the interface. It can simply load some XAML to create the UI and then use F# code for programming user interactions and updating parts of the UI based on change of state (see Program 9). Loading a XAML file for the UI also helps with ensuring that the model, view and controller have been correctly separated when using the Model View Controller design pattern.

There are a number of tools that support generating XAML through a designer interface, including Microsoft Visual Studio, Microsoft Expression Blend and a plugin for Adobe Illustrator, although none of these currently support using WPF with F#. Any XAML created using these tools can still be loaded using F#, with event handlers and other interactions being added manually.

Program 8 A simple XAML file, showing two labels, a textbox and a button in a grid layout.[29]

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="64" />
            <ColumnDefinition Width="128" />
            <ColumnDefinition Width="128" />
            <ColumnDefinition Width="128" />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="24"/>
        </Grid.RowDefinitions>
        <Label Grid.Row="0" Grid.Column="0" >Input: </Label>
        <TextBox Name="input" Grid.Column="1" Text="hello" />
        <Label Name="output" Grid.Row="0" Grid.Column="2" ></Label>
        <Button Name="press" Grid.Column="3" >Press Me</Button>
    </Grid>
</Window>
```

2.4.2 Charts

To display statistics about libraries, it will be necessary to show charts of the data. WPF libraries for displaying charts already exist, so it was not necessary to create a new one.

The first of these libraries that was investigated is the chart control in the WPF Toolkit.[2] An example of a pie chart created using this library can be found in Figure 2.1. This library can create pie charts and bar charts, as well as line graphs, scatter graphs and bubble charts. It is also very configurable, allowing you to set the colours for items in the series and set various properties on the chart itself.

The other chart library that was have looked at for this project is the WpfSimpleChart library.[13] An example of a pie chart created using this library can be found in Figure 2.2. This library can create pie charts, bar charts and stacked bar charts. Unlike with the WPF Toolkit, it is necessary to specify the colours of the items in the series when adding them, rather than allowing them to be automatically set, and it does not appear to be possible to add data in the XAML, instead requiring data binding to be used.

For this project, the WPF Toolkit chart controls were used as they provide a good level of configurability, they look better and can be used with or without the use of data binding. They are also fully supported by the designers in Microsoft Visual Studio and Microsoft Expression Blend.

Program 9 Loading a XAML file and adding an event handler for the button.[29]

```
open System
open System.Collections.Generic
open System.Windows
open System.Windows.Controls
open System.Windows.Markup
open System.Xml

// creates the window and loads the given XAML file into it
let createWindow (file : string) =
    using (XmlReader.Create(file)) (fun stream ->
        (XamlReader.Load(stream) :?> Window))

// create the window object and add event handler to the button control
let window =
    let temp = createWindow "Window1.xaml"
    let press = temp.FindName("press") :?> Button
    let textbox = temp.FindName("input") :?> TextBox
    let label = temp.FindName("output") :?> Label
    press.Click.Add (fun _ -> label.Content <- textbox.Text )
    temp

// run the application
let main() =
    let app = new Application()
    app.Run(window) |> ignore

[<STAThread>]
do main()
```

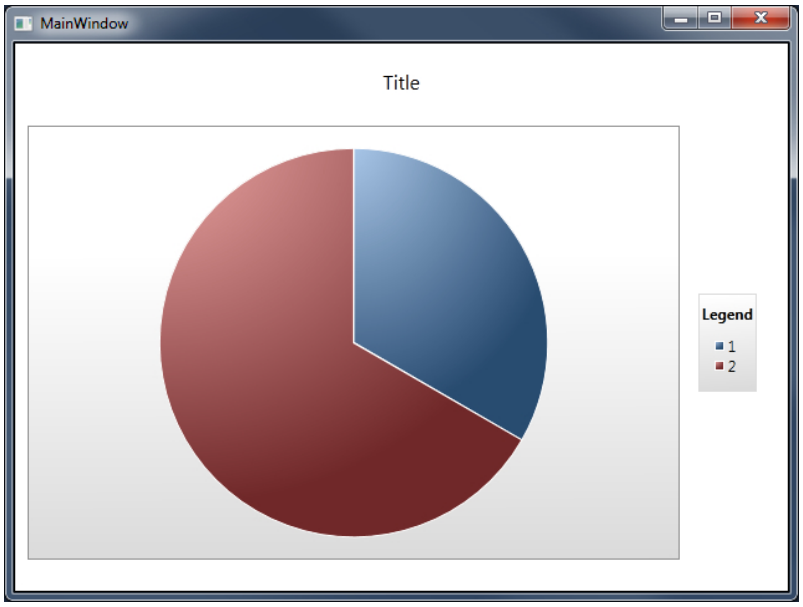

Figure 2.1: Example of a pie chart created with the WPF Toolkit.

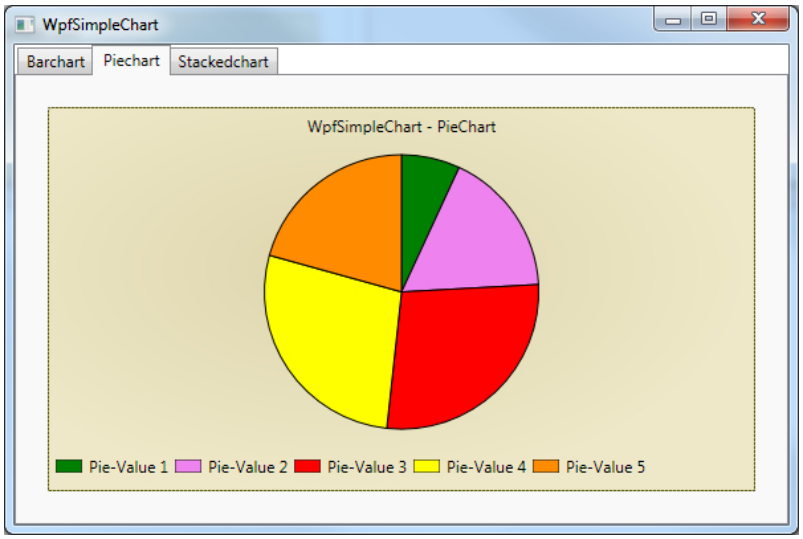

Figure 2.2: Example of a pie chart created with the WpfSimpleChart library.

2.5 Pluggable Architectures

To allow multiple different media libraries to be supported, and new ones to be added without having to rebuild the whole system, the system should use a pluggable architecture to load the media libraries.

With a pluggable architecture, developers can create new dlls which implement interfaces that are exported by the host program, which can simply be "plugged in" to the host to add additional functionality. This solves the problem of "needing to add/remove classes/components from your running system dynamically without any changes to the running code".[12]

There are two pluggable architecture libraries available as standard parts of the .NET Framework: `System.Addin` and the newer Managed Extensibility Framework (MEF). These will each be covered below.

2.5.1 `System.Addin`

For this system to work, the host program must define an interface which defines actions that can be performed with the plugin and will be implemented by the plugin. The host system then "discovers" plugins, for instance by loading dlls from a directory, giving tokens which can then be used to activate and access the plugin. When a plugin is activated, it's functionality is sandboxed to ensure that it can't interfere with the running of the host program, and after this it can be interacted with.[17]

2.5.2 Managed Extensibility Framework

MEF takes a different approach to the use of plugins, where plugins are simply composable parts which are composed to form the final system. As with `System.Addin`, an interface must be defined which will be implemented by the plugin, and the plugin is marked as *exporting* any implementations of this interface. The code that wishes to consume the plugin then *imports* these implementations. A minimal amount of code is then required for finding the plugins and composing the new components with the existing system.[23] An example of using MEF to get plugins can be seen in Programs 10 and 11.

For this project, MEF was used rather than `System.Addin` as it provides all of the features that are required for the system, with a simple and easy to use syntax.

Program 10 An example of a plugin host using MEF[20]

```
module Host
open System.Reflection
open System.IO
open System.ComponentModel.Composition
open System.ComponentModel.Composition.Hosting

type ITastyTreat =
    abstract Description : string

let catalog = new AggregateCatalog()
let directoryCatalog = new DirectoryCatalog(@"C:\Extensions", "*.dll")
let container = new CompositionContainer(catalog)
catalog.Catalogs.Add(directoryCatalog)


type TreatJar() =
    [<ImportMany(typeof<ITastyTreat>)>]
    let cookies : seq<ITastyTreat> = null

    member __.EatTreats() =
        cookies |> Seq.iter(fun tt->printfn "Yum, it was a %s" tt.Description)

let jar = TreatJar()
container.ComposeParts(jar)
jar.EatTreats()
```

Program 11 An example of a plugin using MEF[20]

```
namespace ChocolateCookie
open Host
open System.ComponentModel.Composition

[<Export(typeof<ITastyTreat>)>]
type ChocolateCookie() =
    interface ITastyTreat with
        member __.Description = "Chocolate Cookie"
```

2.6 F# on Other Platforms

Although F# is a language that is designed to run on Microsoft's .NET Framework, this does not mean that F# code can only run on the Windows platform. The following sections discuss the various methods for running F# code on other platforms.

2.6.1 Mono

Novell's Mono project is "an open source, cross-platform, implementation of C# and the CLR that is binary compatible with Microsoft.NET".[25] As such F# code that does not depend on .NET APIs that are not available on Mono can be run on the Mono runtime and all of the core F# libraries are available for Mono.

Although Mono has a fairly complete set of .NET libraries, there are some parts that are not available for Mono. There is currently no support for Windows Presentation Foundation in Mono, and there are currently no plans to add it.[26] Mono does provide other UI technologies such as Windows Forms and Gtk#.[24]

This would mean that the UI would need to be rewritten for a Mono version of the application, however most of the other code could still be used without any compatibility issues.

2.6.2 Android and iOS

Novell have also released tools to allow .NET code to run on Android and iOS devices (Mono for Android and MonoTouch), however F# is not currently a supported language in either of these tools. Some people have successfully run F# code on iOS through MonoTouch[18], however these platforms will not be investigated further through the course of this project.

2.6.3 Windows Phone 7

F# is a fully supported language for use in development on Microsoft's Windows Phone 7 platform. Microsoft's Silverlight framework (a cut down version of the .NET Framework) is used for developing Windows Phone applications, with a UI framework that is very similar to Windows Presentation Foundation.

2.7 Windows Phone 7

As F# is a supported language for Windows Phone development, this is an ideal platform for demonstrating the ability to use F# code across multiple platforms. Many of the libraries available in the .NET Framework for Windows are also available in the Silverlight runtime for Windows Phone, and libraries are available for accessing the various features of the device, from networking to accessing the device's location and playing FM radio.

Much of the code written for the Windows version of this project should compile for Windows Phone without requiring changes, however there are inevitably a few differences between the two, and a completely different style of user interface is required for a mobile device.

2.7.1 Metro

Metro is a set of user interface guidelines published by Microsoft for developing Windows Phone applications, and has a focus on five main principles:[7]

1. Clean, light, open and fast
2. Content, not chrome

3. Integrated hardware and software
4. World-class motion
5. Soulful and alive

These guidelines put more focus on content, with standard typography and user interface controls being used to provide a consistent experience across all applications.

Guidelines are provided on how fonts, icons and controls should be used when developing applications, with a focus on simplicity - it should be obvious what each part of the application does, and all content should be clear and easy to read.

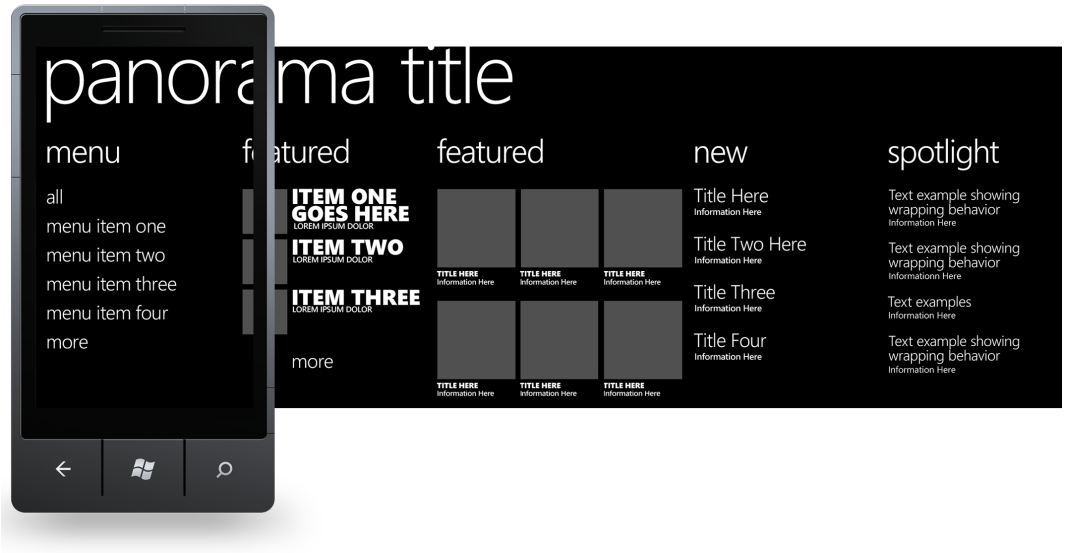

Figure 2.3: The Windows Phone panorama control[7]

As part of the Metro UI, there are some controls available that are specific to applications for mobile devices, for example the panorama control which provides a way of making a single screen of an application scroll horizontally, providing additional space for displaying related material. An example of the panorama control can be seen in Figure 2.3.

2.7.2 Music Library

Windows Phone has a built in library for accessing music library contents from applications. The methods for accessing the library can be found in the `Microsoft.Xna.Framework.Media` namespace, in the `MediaLibrary` class. The various attributes for each item in the library are all available as object attributes, making it easy to access information about tracks. An example of this can be seen in Program 12.

Program 12 An example of accessing track information

```
open Microsoft.Xna.Framework.Media

let lib = new MediaLibrary()

for t in lib.Songs do
    printfn "%s by %s" t.Name t.Artist.Name
```

2.8 Summary

In this chapter, we have seen that there are a number of useful features in the F# language that will help to develop this application, there are existing APIs for accessing information from media libraries and the Internet has a wealth of information about music. These will all allow an application to be created that can provide additional information and statistics about users' local music libraries.

The options available for running F# code on other platforms has also been investigated, going into detail about the Windows Phone platform, covering user interface guidelines and accessing the media library, which will allow a mobile version of the application to be created to demonstrate the cross-platform nature of F# as a language.

Chapter 3

Overall System Architecture

The applications produced as part of this project each have three main sections: the user interface, music library analysis, and using the Internet to access information. These sections are implemented both on Windows as a .NET application, and on Windows Phone 7 as a Silverlight application. Following the overall architecture of the system, the design and implementation of those components that are common to all versions are discussed.

3.1 Project Assemblies

To allow the various parts of the project to be used across multiple platforms, the code of the project was arranged into multiple assemblies, some of which are common to all platforms, and others which are specific to an individual platform. The majority of the code of the project is able to be shared, with only user interface and library interaction code being written for each platform. The main structure of this can be seen in Figure 3.1, with "Common" enclosing those assemblies that can be shared between platforms, "Windows (.NET 4)" enclosing those assemblies that are specific to the Windows implementation, and "Windows Phone 7" enclosing those assemblies that are specific to the Windows Phone 7 implementation. A number of assemblies also exist for testing the various parts of the code base, but as these do not form part of the system, they are not discussed here. The purposes of the different assemblies are discussed below.

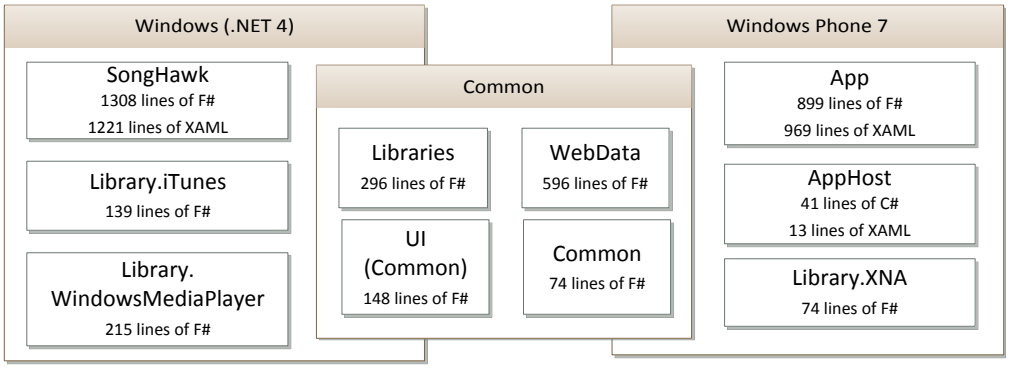

Figure 3.1: Project Assemblies

3.1.1 Common Assemblies

There are four assemblies that are common to both platforms:

- **Libraries:** This assembly provides the interfaces to be implemented by each library implementation, and also provides methods for analysing and ranking the contents of libraries.
- **WebData:** This assembly provides methods to access information from online data sources about genres, artists, albums and record labels.
- **UI (Common):** This assembly provides a class for managing filtering of music libraries, and methods for finding recommendations based on the filtered libraries.
- **Common:** This assembly provides utility functions and types that are used throughout the code base.

3.1.2 Windows (.NET 4) Assemblies

There are three assemblies that are specific to the Windows implementation:

- **SongHawk:** This assembly provides a Windows Presentation Foundation UI implementation, and allows libraries to be "plugged-in" to the application using the Managed Extensibility Framework (MEF).
- **Library.iTunes:** This assembly provides methods for accessing data from an iTunes library by implementing the interface defined in the common Libraries assembly. This assembly is a plugin that is is composed with the application using MEF.
- **Library.WindowsMediaPlayer:** This assembly provides methods for accessing data from a Windows Media Player library by implementing the interface defined in the common Libraries assembly. This assembly is a plugin that is composed with the application using MEF.

3.1.3 Windows Phone 7 Assemblies

There are three assemblies that are specific to the Windows Phone implementation:

- **App:** This assembly provides the user interface implementation for Windows Phone, and brings together the various components of the system.
- **AppHost:** This assembly provides the application information that is required for a Windows Phone application, but does not provide any implementation of the system, instead calling the App assembly. This assembly uses C# rather than F#, as the Windows Phone development tools are unable to deploy purely F# applications.
- **Library.XNA:** This assembly provides methods for accessing data from the XNA library system that is used on Windows Phone 7 by implementing the interface defined in the common Libraries assembly. As there is only one type of library available on Windows Phone, this assembly is directly incorporated into the system without the use of MEF.

3.2 Third Party Libraries

Although most of the code for this project was implemented from scratch to provide functionality specific to the applications, some third party libraries were used to provide additional functionality.

3.2.1 MusicBrainz Sharp

The MusicBrainz Sharp library (see Section 2.3.3) was used to provide access to the MusicBrainz data source rather than writing a new wrapper for the MusicBrainz web service. This library is used to provide various dates relating to artists, as well as the record labels that an album was released by.

3.2.2 Parsing of Web Request Responses

Json.NET[19] is a library for reading and writing JSON (JavaScript Object Notation) encoded data, such as that used for Freebase queries. This library is available for Windows Phone as well as the full .NET Framework and as such is used across both versions of the application.

The Last.fm and Spotify web services return data encoded as XML, which can easily be parsed using the XML processing functionality that is built into the .NET Framework. This did not require any additional libraries to be included, however some helper functions were created to aid in accessing data from these responses.

3.2.3 Charts

The WPF Toolkit[2] was used to allow pie charts to be displayed in the Windows version of the application as described in Section 2.4.2.

The Silverlight Toolkit[1] provides the same functionality for the Windows Phone version of the application.

3.2.4 Music Library Access

For the Windows version of the application, the COM (Common Object Model) libraries for Windows Media Player and iTunes were used to provide access to these media libraries (see Section 2.2). For the Windows Phone version, access to the music library is available through a standard assembly (see Section 2.7.2).

3.3 Component Composition

The user interface for both versions of the application use the Model View Controller design pattern due to the way that the user interface libraries on each platform function. This helpfully allows the different components of the application to be composed in the controller layer of the application.

3.3.1 Model

The model of the application is made up of the various common libraries providing access to web data and the library analysis functions, as well as the music library plugins.

The library plugins for Windows are not a core part of the application, and are instead "plugged-in" to the system using the Managed Extensibility Framework that is a core part of the .NET Framework (see Section 2.5.2), allowing additional libraries to be added without needing to make changes to the application as a whole.

For the Windows Phone version of the application, this is not possible, due to the lack of the Managed Extensibility Framework, however it is also unnecessary, as only access to the core media library is needed.

3.3.2 View

On both platforms, the view is a collection of XAML (eXtensible Application Markup Language) files, which define the layout of the application. This is populated with data from the model via the controller, often using data binding to remove the need to create controls in the controller.

This layer of the application does not consist of F# code, as it only provides the UI layout, not adding any functionality of it's own.

3.3.3 Controller

The controller provides the functionality to the various parts of the view, and also populates the view with data from the model. This layer of the application brings all of the components together, removing the need for the view to have any information about the model, and vice versa.

The controller functionality is contained within the `SongHawk` and `App` assemblies in the Windows and Windows Phone versions of the application respectively.

3.4 Music Library Abstraction

Due to the different ways of accessing data from different music libraries, for example the use of strings to access attributes with Windows Media Player and the use of object attributes for iTunes, an abstraction layer was created to provide a standard interface for accessing the contents of libraries. This also allows additional libraries to be added in future without having to make any changes to the analysis and data retrieval code.

This abstraction layer supports the following operations:

- **GetName:** this returns the name of the library and is used to populate lists showing which libraries are available and also provides a unique identifier for library selection.
- **GetGenres:** this returns a collection of genre names as strings, which are used throughout the application, as well as providing keys for filtering data accessed through other library methods.
- **GetArtists:** this returns a collection of artist names as strings, which are used throughout the application, as well as providing keys for filtering data accessed through other library methods. This method also takes a list of genres, allowing the results to be filtered to only those artists that are categorised by the those genres.
- **GetAlbums:** this returns a collection of album names as strings, which are used throughout the application, as well as providing keys for filtering data accessed through other library methods. This method also takes a list of genres and a list of artists, allowing the results to be filtered to only those albums that are categorised by the those genres or released by those artists.
- **GetTracks:** this returns a collection of tracks, which are used in various parts of the application. Tracks are represented as an object with a Title, Length, Genre, Album, Artist, Playcount and Year. This method takes a list of genres, artists and albums, allowing the results to be filtered to only tracks on a set of albums, by a set of artists or that are from a set of genres, or any combination of these.
- **IsAvailable:** this returns a boolean representing whether a plugin is compatible with a given system, for instance the iTunes plugin returns false if iTunes is not installed.

The class structure for this section can be seen in Figure 3.2, and this interface is exposed to be implemented in other assemblies and connected using MEF.

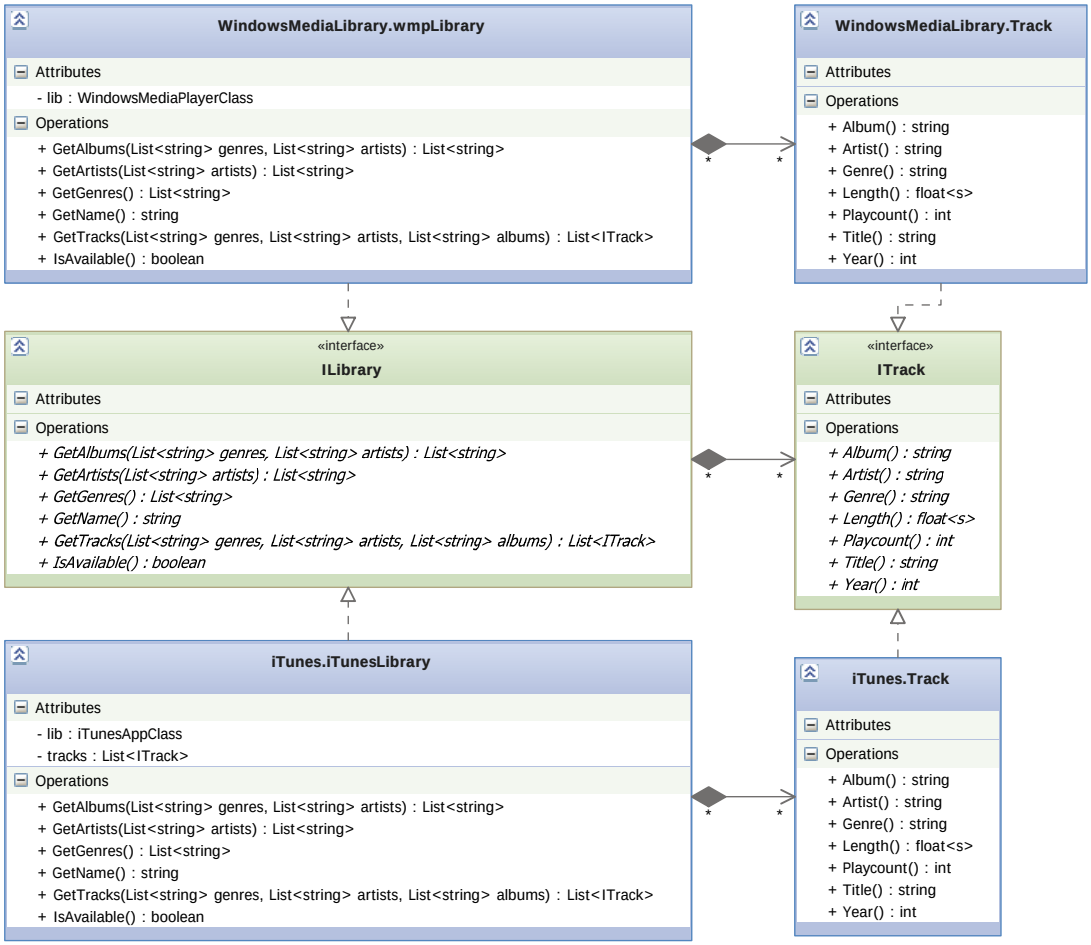

Figure 3.2: Class diagram for the library abstraction layer

3.5 Music Library Analysis

This section of the project also concerns analysing data from music libraries to allow statistics to be shown in the UI. This is done by providing a number of methods which extract data from a given collection (as provided by the library abstraction layer) and return a set of key-value pairs which can then be returned to the UI layer for rendering.

The statistics that are supported by this part of the system are:

- Average and total play counts for tracks, artists, albums or genres
- Average and total song counts for artists, albums or genres
- Average and total durations for artists, albums or genres
- Years for tracks, artists, albums or genres

3.5.1 Rankings

Part of the analysis provided by the application is the ability to rank library contents based on play counts, song counts, and total durations of tracks. These rankings are provided for genre, artist and album and are calculated by:

1. Constructing lists of the various attributes required by iterating over the results from a library query.

2. Normalising the values using the standard score method:

$$Z = \frac{X - \mu}{\sigma}$$

where X is the raw score to be normalised, μ is the mean of the set of raw values, and σ is the standard deviation of the set of raw values.

3. Weighting the different attribute values, with song counts, durations and play counts having a ratio of 1:1:2.

4. Totalling the weighted values and sorting the list to find the top items for each category.

The functional programming paradigm and list processing functions of F# make this an ideal language for calculating such statistics, with method chaining being used to pass data between the various steps of the algorithm.

3.5.2 Recommendations

Recommendations of artists is another part of the analysis section of the project, with recommendations being made based on play counts, song counts and total durations. The artists are selected by ranking the artists by each of the criteria, then taking the similar artists from Last.fm for the top 10 library artists. Artists that are already in the local library are then removed from the list, and artists are then sorted based on the number of times they have appeared in the list.

3.6 Web Data

This part of the project involves getting data from various web services based on criteria passed in, such as the selected album or artist.

For most requests to these web services, the responses are cached to reduce the amount of requests being made to the services. This helps to prevent any usage limits being reached and also improves responsiveness of the system as data can be found from a local cache rather than making a HTTP request.

3.6.1 Last.fm

The main use of the Last.fm API is to get biographical information about artists and additional album information. It is also used for getting track listings for albums, finding the albums released by an artist and finding similar artists. The following methods are provided for accessing Last.fm data:

- Get artist biography
- Get album information
- Get albums by artist
- Get track listing for album
- Get similar artists
- Get image for artist
- Get album artwork

These methods use basic HTTP requests to get the data from the API and then use the built in XML parsing methods to extract the relevant data from the responses to be returned to the requesting code in the required data type.

3.6.2 MusicBrainz

The MusicBrainz API is used to find additional information about artists and albums, including relevant dates for the artist (birth and death dates for individuals or founding and dissolving dates for groups). It is also be used to provide information on record labels that an artist or release is associated with.

These methods use the MusicBrainz Sharp library to remove the need to write code for interacting with the XML Web Service and the following methods are available:

- Get artist dates
- Get record label information (for release or for artist)

3.6.3 Freebase

The Freebase web service is used to get additional information about record labels, artists and albums to supplement biographical data from Last.fm. This includes finding other artists that are signed to the same record label and getting information about the record label, as well as finding information about genres.

The following methods are available:

- Get record label artists
- Get record label information
- Get genre information
- Get genre artists

Freebase requests are generated by encoding the request as JSON and sending that as a query string parameter on a HTTP request, with the response being returned as JSON before the relevant information is extracted using the Json.NET library.

3.6.4 Spotify

The Spotify Metadata API is used to get Spotify URLs to load tracks from the currently selected artist or album in Spotify. The user is also given the option to use a HTTP URL rather than a Spotify URL if they would prefer to share the URL with other people or do not have Spotify installed.

Methods are available to return Spotify URLs for artist, album or track and also to convert a Spotify URL to a HTTP URL.

3.7 Summary

This chapter has introduced the various parts of the system, including third party libraries, and described how they interact with each other to form the completed applications. The various common components of the system have also been introduced, with details on how music libraries are accessed and analysed, and what data is retrieved from the Internet to be provided to the user. The following chapters will go into further detail about the structure and implementation of the components for specific versions of the application.

Chapter 4

Windows Application

This chapter discusses the various aspects of the Windows version of the application, specifically the user interface design and the music library plugins.

4.1 UI Design

As mentioned earlier, the user interface follows the Model View Controller design pattern. The model is used to provide the data to the application, so is not discussed in this section, however the view and controller are integral parts, defining the layout of the application, and selecting what data is to be included in each area of the application. In this section, the view and controller will be considered together, as they form the application that the user sees.

4.1.1 Overall Design

A simple approach was taken to the design of the Windows version of the application, with icons being used to provide easily recognisable links to the parts of the application, and text mainly being used to provide information to the user, rather than distract the user from the content.

There are no complex menu structures, with most parts of the system being accessible from the main screen, and apart from the library selection and navigation history, there is very little state to maintain throughout the application.

The library filtering information, navigation and settings are all accessible from any part of the application, sitting at the top of the window and at the left to ensure that the user can always find their way to a different part of the system.

The bar at the top of the window provides the user with ways of navigating around the application, as well as selecting their choice of music library. The bar at the left of the window shows the number of genres, artists and albums that are included in the current dataset (initially the entire contents of the library), and provides a button to access a screen to modify the dataset to only include the parts of the library that the user cares about.

4.1.2 Main Screen

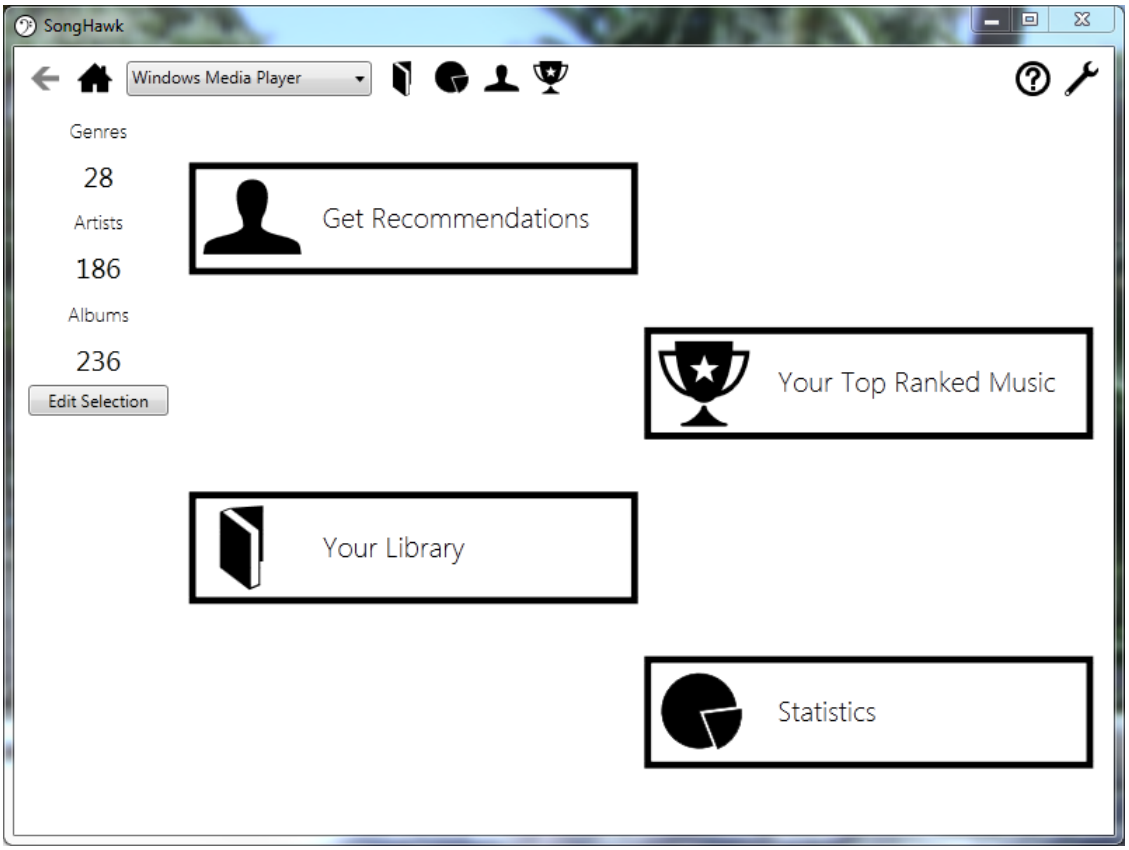

Figure 4.1: The main screen

As shown in Figure 4.1, the main screen provides access to all of the four main areas of the application:

- Recommendations
- Rankings
- Library Contents
- Statistics

No other information is provided on this screen, allowing the user to jump to the parts of the application that they want to use without having information that they may not want to see being shown.

Due to the simple layout of the main screen, the controller only needs to attach events to the buttons on this screen to allow navigation to occur, however when the application is loaded the library data is also loaded into the system. After the application has been loaded, this data is available from all screens in the application and it is not necessary to reload the data unless the library source is changed.

4.1.3 Statistics

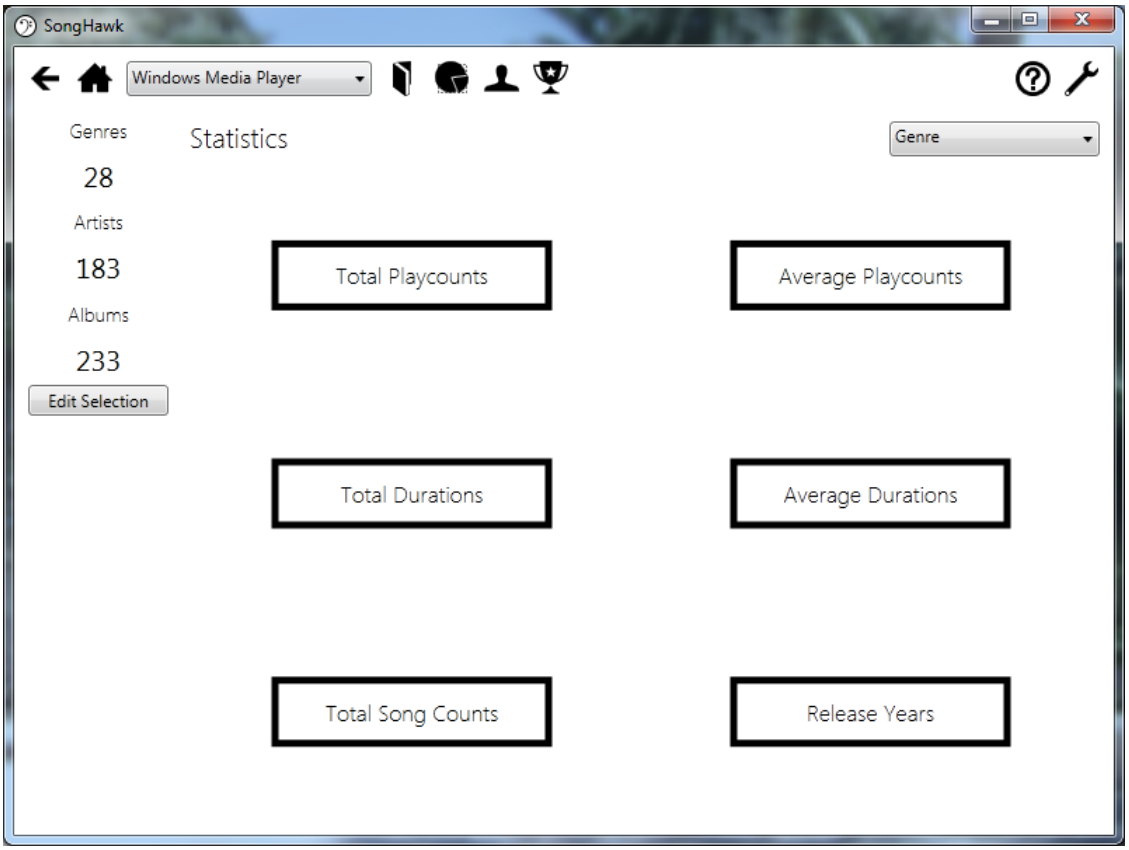

Figure 4.2: The statistics screen

As shown in Figure 4.2, the statistics screen allows the user to choose between the various statistics that are available. Four different levels of statistics are available: Genre, Artist, Album and Track, and this can be selected from the combo box in the top right of the screen. The user can then choose between the following types of statistics:

- Total Playcounts
- Average Playcounts
- Total Durations
- Average Durations
- Total Song Counts
- Release Years

The controller then loads the relevant data using the analytical requests in the model, feeding the results into a pie or stacked column chart depending on the option chosen. Examples of these can be seen in Figures 4.3 and 4.4. Due to the need to aggregate information from the selected parts of the library, loading statistics screens can take a large amount of time.

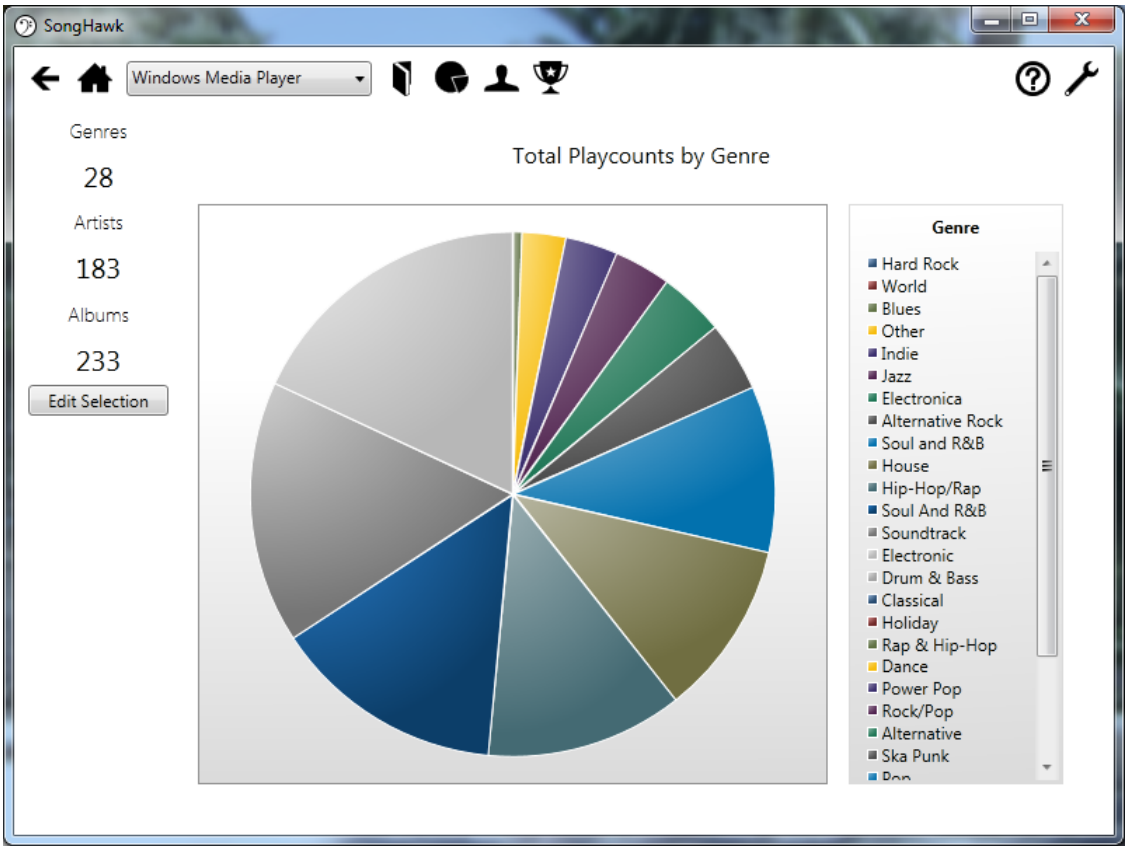

Figure 4.3: Total Playcounts by Genre

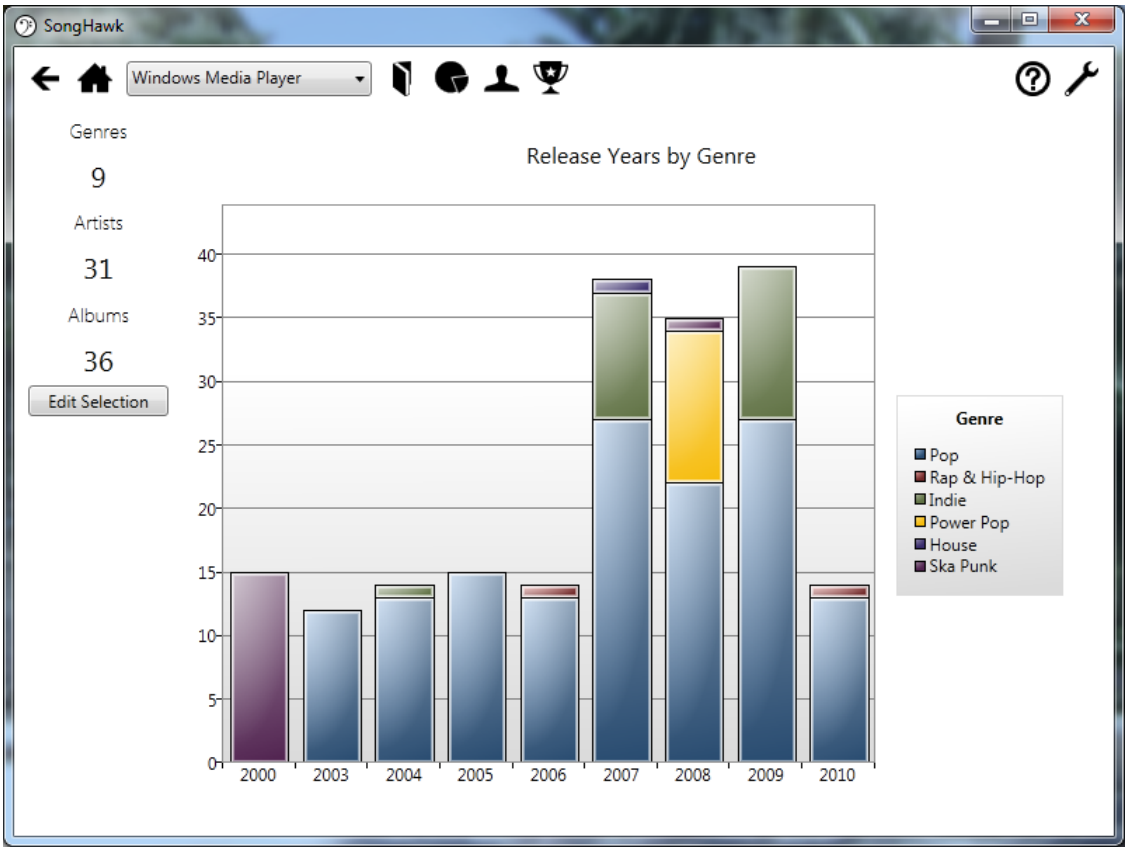

Figure 4.4: Release Years by Genre

4.1.4 Library Contents

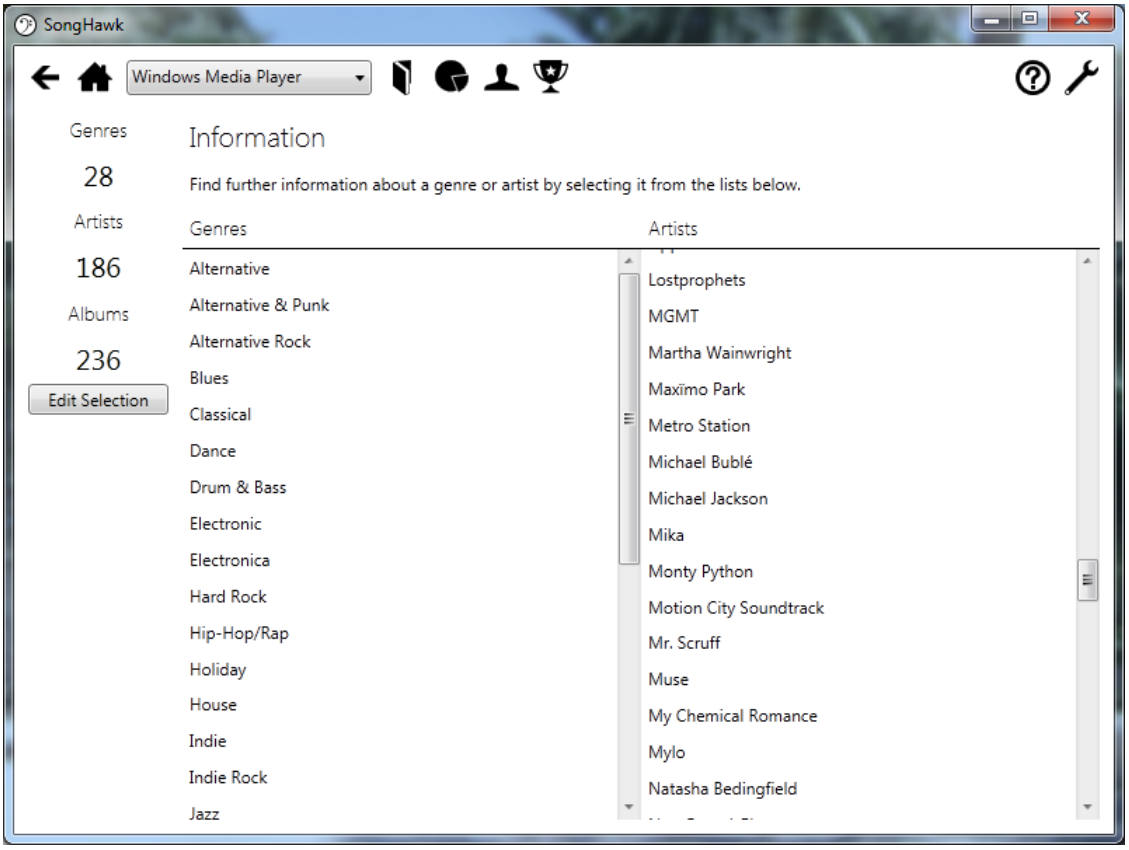

Figure 4.5: Library contents screen

To allow the user to easily access information about the genres and artists in their library, this screen shows lists of these, from which the relevant information pages can be loaded as shown in Figure 4.5.

The controller gets lists of genres and artists contained within the library and adds these to the view, attaching handlers to the click events. As the F# compiler requires that files be compiled in a given order, a global mutable variable is used to store loaders for the various types of information page, with this variable being set to a record containing the different load functions when the application is launched.

4.1.5 Library Filtering

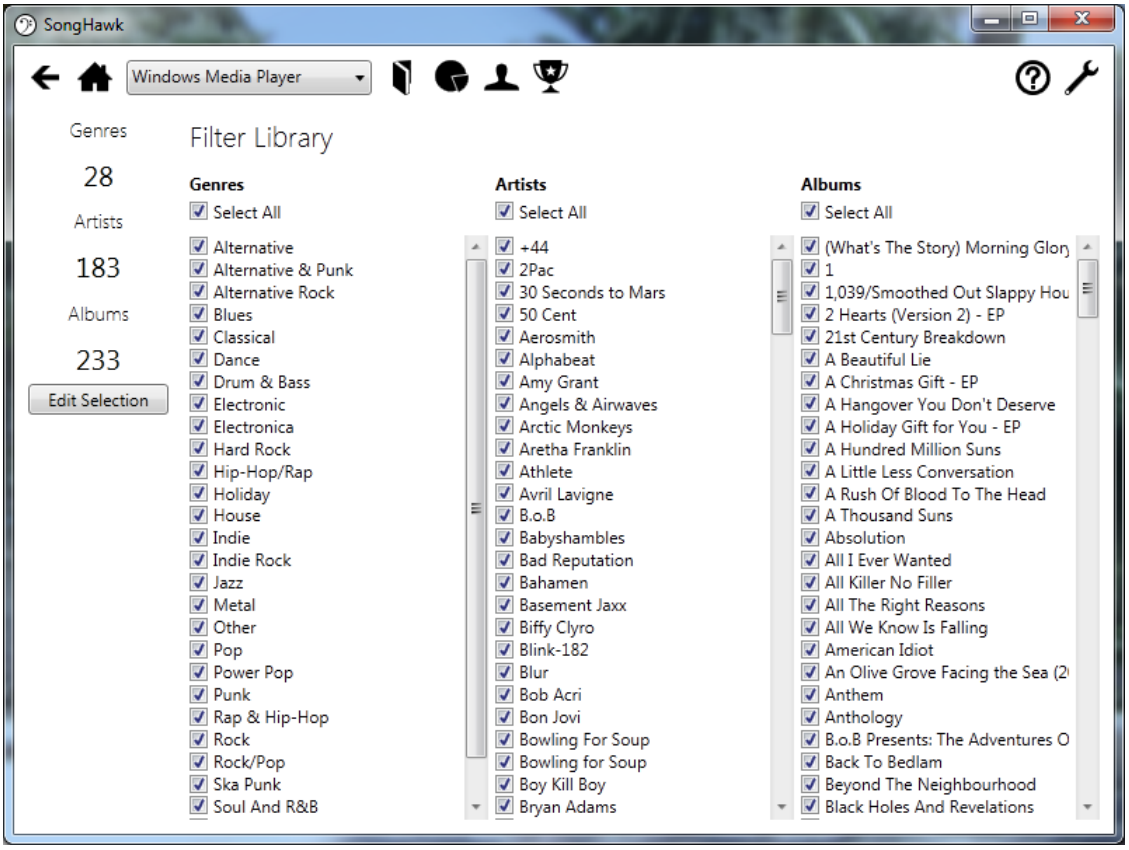

Figure 4.6: Library filtering screen

The user is provided with the ability to restrict the library contents to only show those items that they are interested in. This is done using the library filtering screen (Figure 4.6) and the selection is then used across all areas of the application.

The selection is used throughout the application through the use of three global variables storing instances of the `LibrarySelectionManager` class, which stores the available and selected library items and allows the list to be filtered using a passed in filtering function. As the item selections are changed, the items that are visible in the other lists are also modified to only show those items that are still applicable.

4.1.6 Genre Information

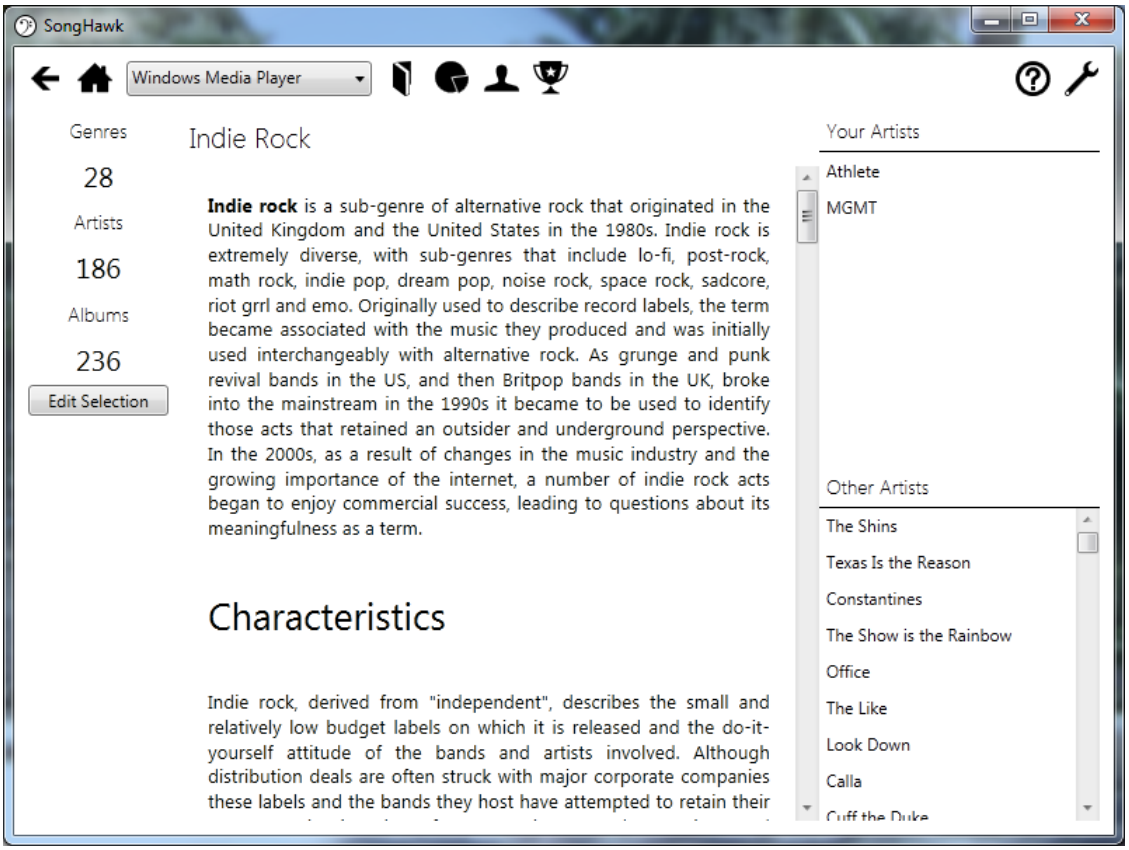

Figure 4.7: The Genre Information Screen

The genre information screen (shown in Figure 4.7) provides an overview of the genre (taken from Freebase), along with a list of artists from the user's library that have music classified by this genre and also a list of other artists of this genre (again taken from Freebase).

The information about the genre is processed to convert it from an HTML document to a XAML flow document, with any unsupported HTML tags being stripped out prior to rendering. The generated flow document is then styled to make the fonts fit in with the rest of the application.

4.1.7 Artist Information

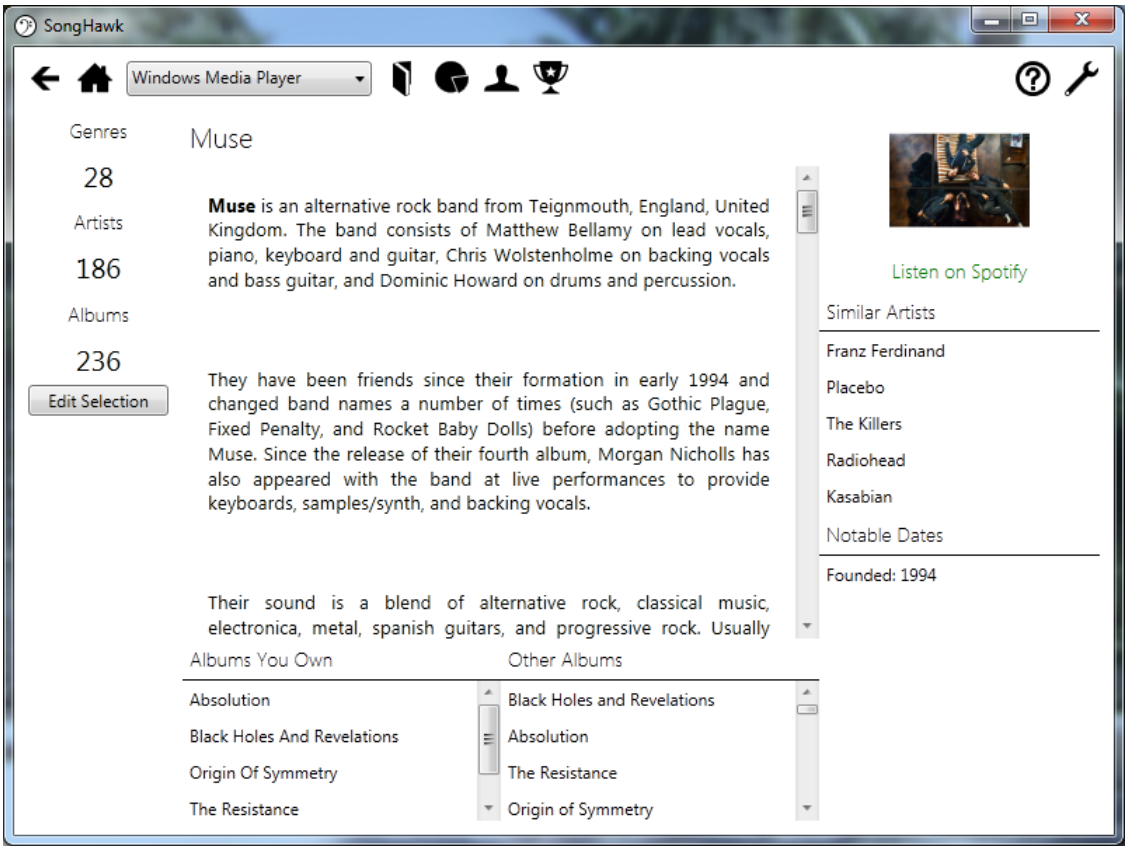

Figure 4.8: The Artist Information Screen

The artist information screen (shown in Figure 4.8) provides a biography of the artist, along with relevant dates (e.g. when born/died), a list of similar artists (found from Last.fm) and lists of albums (both owned and not owned).

As with the genre information screen, the information about the artist is processed to make it suitable for displaying in the application, and if possible, a link is provided to navigate to the artist in Spotify, to allow users to listen to music by this artist, whether or not they actually own any albums by them. If Spotify is not installed, or the user has changed their options, this link will load the Spotify launch page for this artist in the browser, giving links to install Spotify, or simply allowing the user to get a URL to send to other people.

4.1.8 Album Information

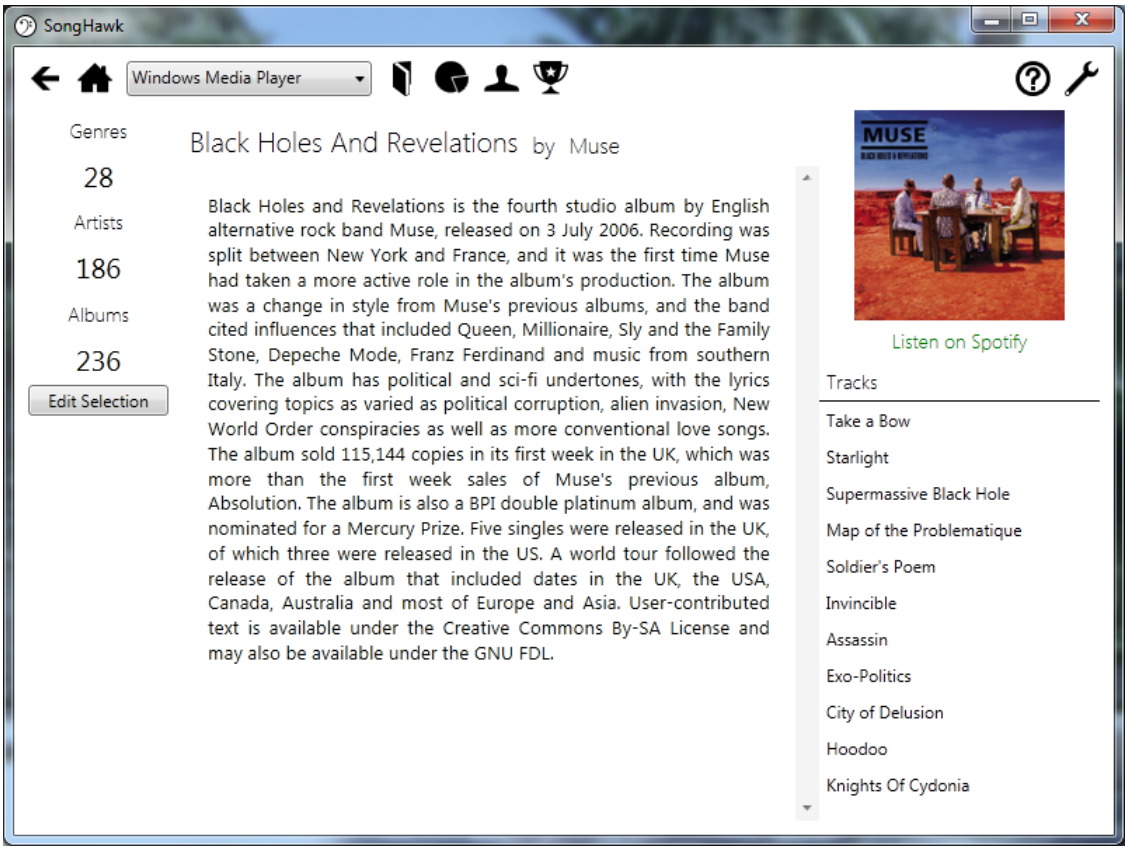

Figure 4.9: The Album Information Screen

The album information page (shown in Figure 4.9) provides a description of the album, along with artwork and a track listing. The information that is used to populate all areas of this screen is found from Last.fm and as with the artist information screen, a link is provided to play the album in Spotify.

The information about the album is also converted from HTML to a XAML flow document and there are no additional pages accessible from this screen.

4.1.9 Recommendations

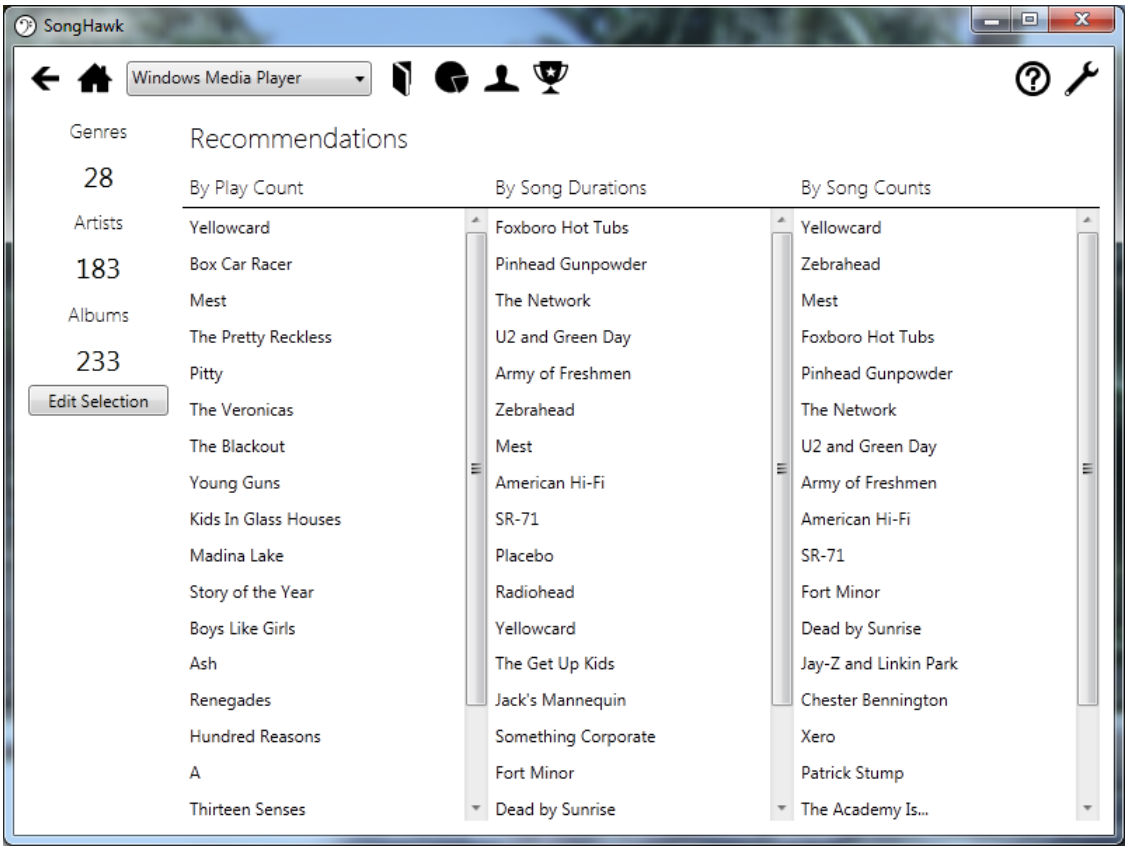

Figure 4.10: The Recommendations Screen

The recommendations screen (shown in Figure 4.10) shows recommended artists as described in Section 3.5.2. The recommendations are found based on the user's current selection and then the results are added to the screen, with click handlers being added to allow the user to navigate to each of the artist information pages.

4.1.10 Rankings

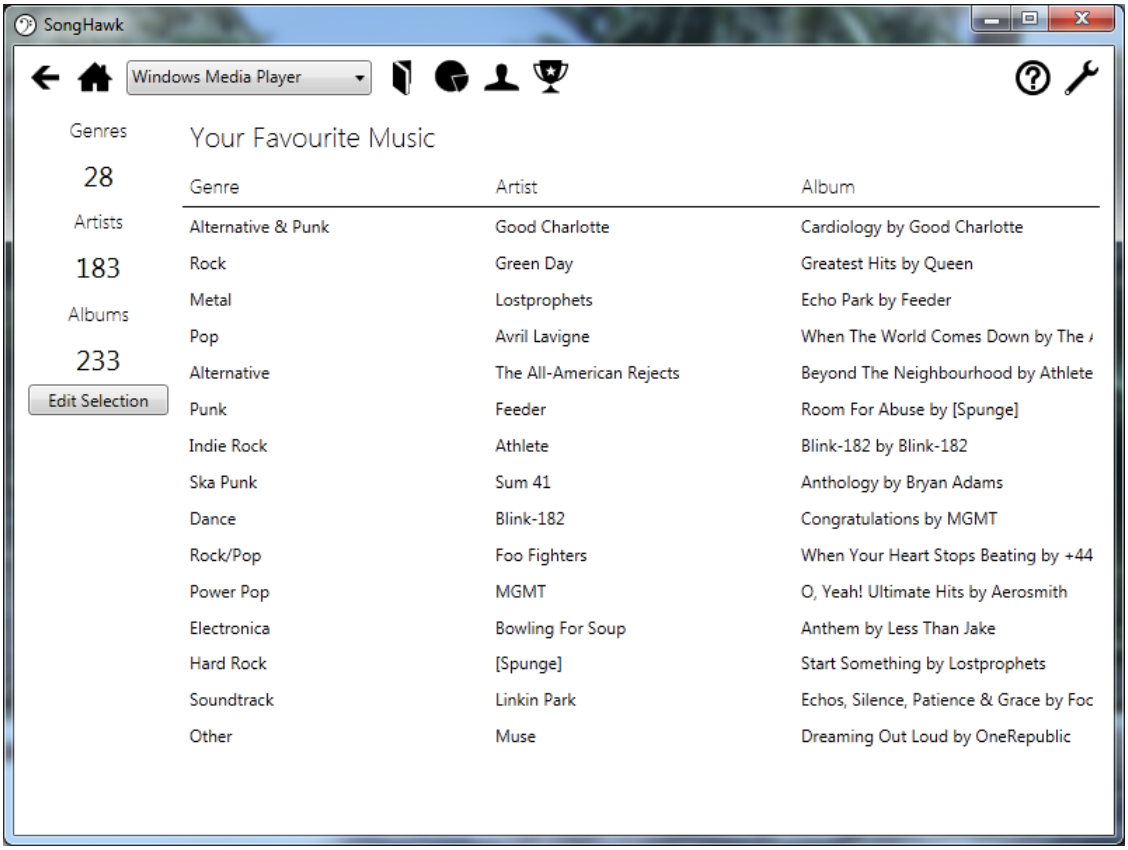

Figure 4.11: The Rankings Screen

The rankings screen (shown in Figure 4.11) shows the top ranked music in the library, for genres, albums and artists, by the method described in Section 3.5.1. As with the recommendations screen, these results are added to the screen, with click handlers added to allow the user to go to the relevant information pages.

These results are always for the library as a whole, not taking into account the library filtering that the user has put in place. This is to simplify the underlying code and make it easier to use across all types of library and platform.

4.1.11 Options Screen

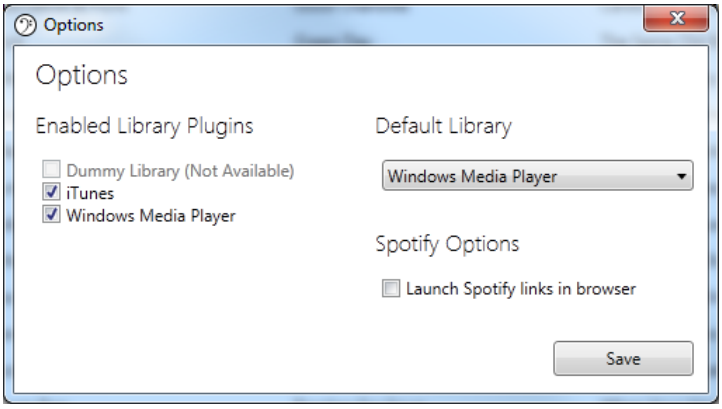

Figure 4.12: The Options Screen

The options screen (shown in Figure 4.12) allows the user to specify which library plugins are enabled (and which should be the default), as well as choosing whether Spotify URIs are launched in Spotify or the browser (if Spotify is not installed, these will always launch in the browser). These options are provided through the use of an `ApplicationSettings` class which allows the .NET Framework's built in settings manager to be used without having to decide how settings should be stored on the user's system.

This uses an object-oriented approach rather than a functional approach, but due to F# being a multi-paradigm language, this does not present a problem, it is as easy to write object-oriented code in F# as it is to write functional code.

4.1.12 Help/About Screen

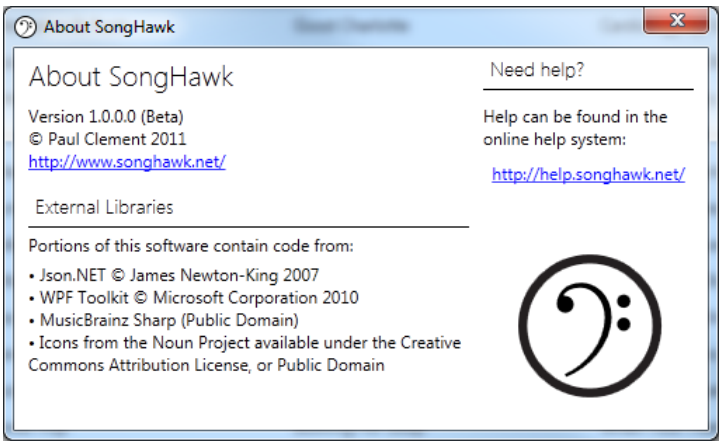

Figure 4.13: The About Screen

This screen (shown in Figure 4.13) allows the user to find out basic information about the application and also provides a link to online help to allow the user to find additional information about how the application should be used. It was decided not to include the help system as part of the application, as this would not allow additional topics to be added to the help system without distributing an update which includes the updated information.

4.2 Windows Media Player Plugin

The Windows version of the application has a plugin for Windows Media Player to allow library information to be loaded. This plugin is a separate assembly, which is incorporated into the application using the Managed Extensibility Framework. To do this, it implements the `SongHawk.Libraries.ILibrary` interface, providing the methods required for accessing the data.

It accesses the data through the Windows Media Player COM API, which also supports querying on attributes such as genre or artists. As such, it is not necessary to directly filter data within the F# code, however it was necessary to create some helper functions to convert string lists into the required form for passing to the API. This was done through the use of a structure of lists, representing parameter groups, the individual parameters in each group, and the parameter forming the condition, as shown in Program 13. To assist in this, additional helper functions where added to build the parameter lists to pass into this function.

4.3 iTunes Plugin

An iTunes plugin is also available, which uses the iTunes COM API to access data from the iTunes library. Unlike the Windows Media Player API, the iTunes API does not support filtering data

Program 13 Method for adding query parameters for Windows Media Player

```
let internal AddParameters (query : IWMPQuery) (parameters : string list list list) =
    for paramGroup in parameters do
        for [attribute ; operator ; value] in paramGroup do
            query.addCondition(attribute, operator, value)

        query.beginNextGroup()
```

and also has a comparatively slow access speed. To address this, the library is cached within the application at load time and data selection is then performed on this list of tracks.

When loading the tracks into the cache, non-audio tracks (e.g. videos, iOS applications) are removed from the selection and then a `SongHawk.Libraries.ITrack` object is built for each item. This is stored as an F# sequence, which is passed through `Seq.cache` to ensure that it is only loaded once. A number of functions are available for use with `Seq.filter` to filter the data as requested, and these are shown in Program 14. These use F# Active Patterns to extract the genre, artist or album name from an `ITrack` when the function is defined, rather than having to extract the data within the function.

Program 14 iTunes library filtering methods

```
let internal HasGenre (genres:string seq) (Genre genre)  =
    match genres with
    | SeqEmpty -> true
    | _ -> (genres |> Seq.exists (fun g -> g = genre))

let internal HasArtist (artists:string seq) (Artist artist) =
    match artists with
    | SeqEmpty -> true
    | _ -> (artists |> Seq.exists (fun a -> a = artist))

let internal HasAlbum (albums:string seq) (Album album) =
    match albums with
    | SeqEmpty -> true
    | _ -> (albums |> Seq.exists (fun a -> a = album))
```

4.4 Summary

In this chapter, the various screens of the application and their underlying implementation were discussed, along with the implementation details for the two library plugins that are provided for this version of the application. The different techniques used to display data were discussed, as well as what information is provided to the user in each part of the application.

Chapter 5

Windows Phone 7 Application

The Windows Phone 7 version of the application draws strongly on the Windows version, with much of the same underlying logic being used to generate the user interface. Despite this, it was necessary to build a new user interface which is more suitable to the mobile form factor, and also to integrate with the Windows Phone media library rather than providing a plugin system.

5.1 UI Design

The UI design for the Windows Phone application is heavily inspired by the Metro guidelines described in Section 2.7.1, with a simple layout where the content is made as clear as possible. There are some parts of the Windows application that are not available in the Windows Phone application, such as Spotify links and data retrieved from MusicBrainz, however most features are still available. There are also no configuration options, as there is only one library source available. The application mainly uses white text and graphics on a black background, with additional colours only being used for artist/album artwork and segments of pie charts. This helps to draw the user to the content of the application.

The application also makes use of two Windows Phone user interface controls: the pivot and panorama controls. These both allow the application to provide more information than will initially fit on the screen, with the pivot control providing an interface that is similar to a tabbed interface, and the panorama control allowing the screen to extend horizontally, allowing the user to swipe across to access further information. Both of these controls allow the user to swipe across using their finger, and wrap around so it is not necessary to swipe back through the screen to get to the first panel. The pivot control is used for lists of data, whereas the panorama control is used on the information pages for genres, artists and albums.

Unlike the Windows application, where there are no underlying classes for each of the XAML files forming the view, the Windows Phone version does have a class for each screen, allowing the navigation features of Windows Phone to be used: pages are loaded by referencing a XAML file rather than by calling a method, and this in turn initialises the relevant class.

5.1.1 Main Screen

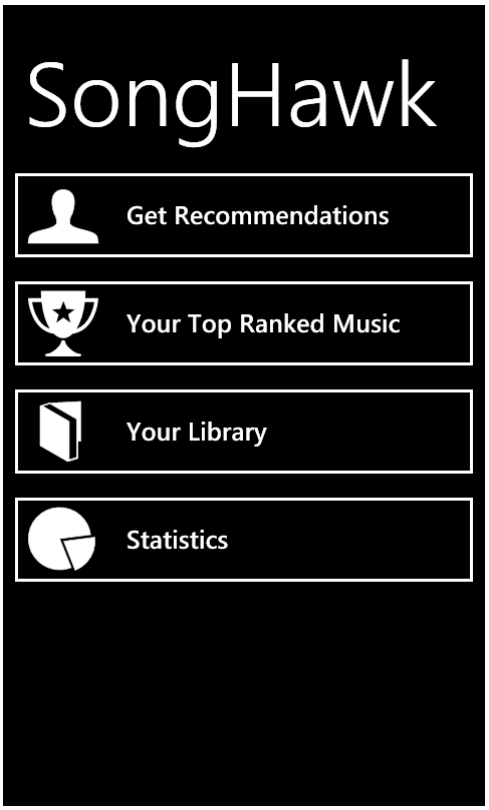

Figure 5.1: Windows Phone Main Screen

The main screen (Figure 5.1) provides easy access to the four main sections of the application: recommendations, rankings, library contents and statistics. This is presented as a series of buttons, each containing an icon and a textual representation of the feature.

The main difference between this screen and the window of the Windows version of the application is that it is not possible to filter the library to only use a subset. This is to keep the mobile version of the application simple and reduce the amount of memory that the application needs to store data.

5.1.2 Recommendations

Figure 5.2: Recommendations

This screen (Figure 5.2) shows recommended artists based on playcount, song durations and song counts of the existing library, displaying the results in a pivot control to allow the user to swipe across the screen to get to the different lists.

The list of artist names is populated using data binding on a list, with the user being able to press each artist name to be taken to the relevant information page.

5.1.3 Rankings

Figure 5.3: Rankings

This screen (Figure 5.3) shows the top ranked genres, albums and artists in the library, again displaying them in a pivot control to allow the user to swipe across the screen to get to the different lists.

As with the recommendations screen, the lists are populated using data binding, with event handlers having been added to allow the user to press a title and be taken to the relevant information page.

5.1.4 Library

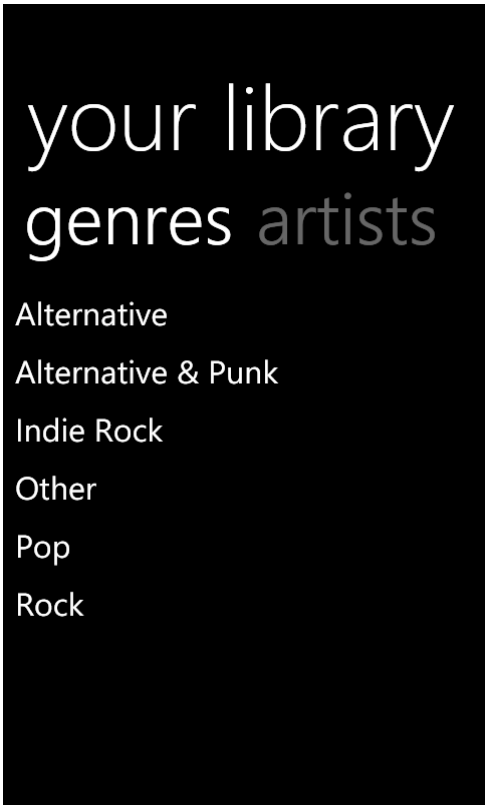

Figure 5.4: Library Contents

This screen (Figure 5.4) shows the genres and artists currently in the library in a pivot control. This allows the user to easily access information about the contents of their library.

The techniques used for displaying data and the methods of user interaction available from this screen are the same as with the rankings screen, with the user being able to access all of the relevant information screens by pressing on the various list items.

5.1.5 Genre Information

Figure 5.5: Genre Information

This screen (Figure 5.5) provides information about a given genre, including a description, list of artists in the user's library and a list of other artists that fit that genre. This information is displayed in a panorama control, with the user swiping across to find more information. As with most of the lists in the application, the user can press on list items to be taken to the relevant information page.

The information about the genre is not processed in the same way as in the Windows version. Rather than converting the HTML to a XAML flow document, the text is split into paragraphs and then all HTML tags are stripped from the data. The information is then displayed by adding each paragraph as a `TextBlock` to a `StackPanel`. This helps to keep the formatting relatively simple for display on a smaller screen.

5.1.6 Artist Information

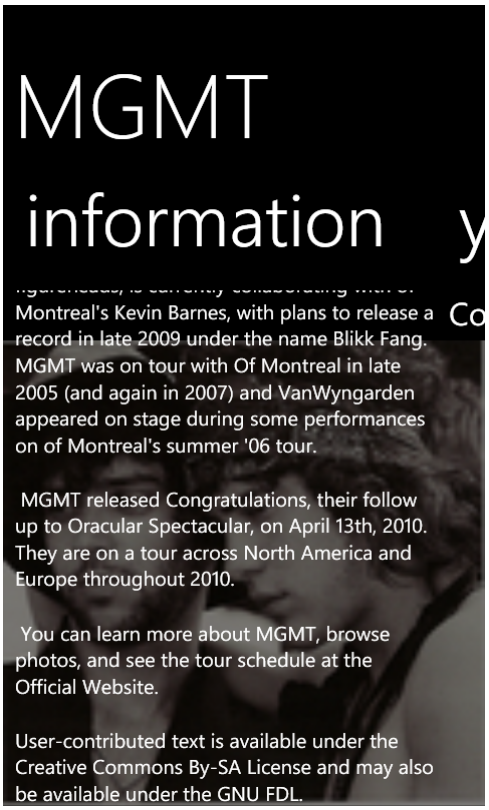

Figure 5.6: Artist Information

This screen (Figure 5.6) provides information about a given artist, including a biography, albums (both owned and not), and similar artists. An image representing the artist (found from Last.fm) is shown in the background, with the information being shown in a panorama control. The user can press any of the similar artist names or album titles to be taken to the relevant information page.

This data is populated in much the same way as the genre information screen, again deviating from the approach taken for the Windows version of the application.

5.1.7 Album Information

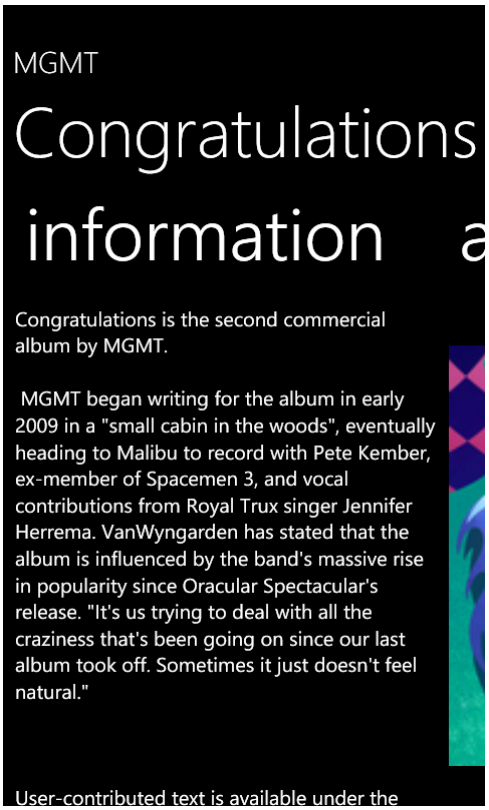

Figure 5.7: Album Information

This screen (Figure 5.7) provides information about an album, including information, album artwork and a track listing. Like the other detail pages, this information is shown in a panorama control. Unlike the artist information screen, the album artwork is displayed on a panel of it's own rather than being in the background, as this allows the user to see the image clearly and is an integral part of the album.

The implementation of this screen is similar to the artist and album information screens, using data binding and processing of the information for displaying on this type of device.

5.1.8 Statistics

This screen (Figure 5.8) allows the user to access statistics about their music library. The statistics are available for genres, artists or albums, with the following statistics available:

- Total playcounts
- Average playcounts
- Total durations
- Average durations
- Total song counts

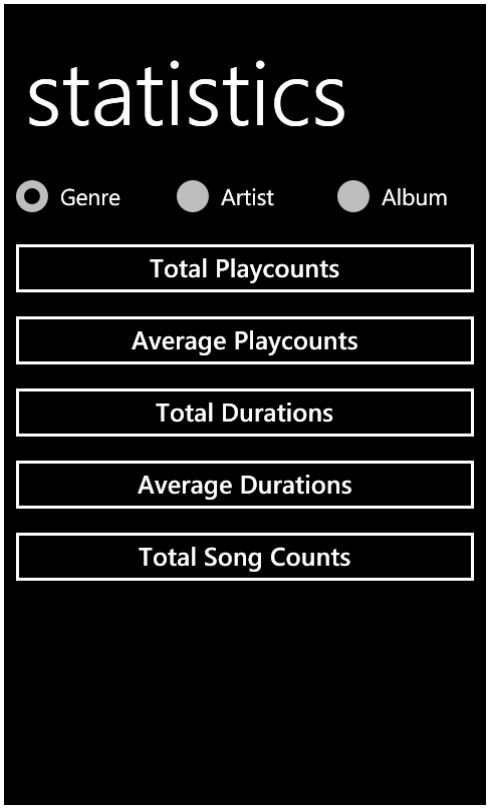

Figure 5.8: Statistics Screen

Each of the statistics are shown as a pie chart (see Figure 5.9), and when a pie chart segment is pressed, the application navigates to the information page relating to that segment. The pie charts are part of the Silverlight Toolkit for Windows Phone, and get their data through the use of data binding to the results returned from the analysis functions.

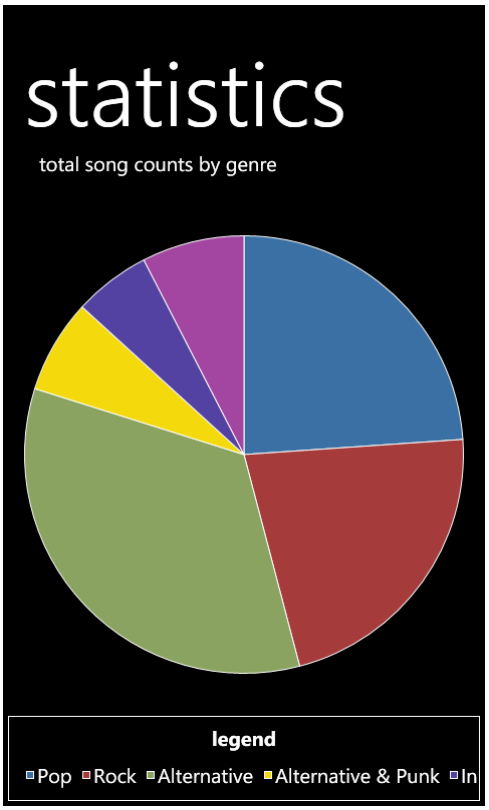

Figure 5.9: Total Song Counts by Genre

5.2 Windows Phone Music Library

Accessing data from the Windows Phone Music Library is achieved as described in Section 2.7.2, through the methods in the `Microsoft.Xna.Framework.Media` namespace, with the various items of data in the library being accessible through object methods.

Although methods exist to access lists of Genres, Albums, Artists and Songs in the library, there are no built in query operations to allow this data to be filtered. As such, filtering is applied through the code when the items are requested.

5.3 Issues Encountered

While in general the code that had been written for the Windows version of the application was fully compatible with the Windows Phone 7 version, there were a few areas where changes needed to be made:

- The `System.Net.WebClient` class, which is used to download data over HTTP, does not have the same methods on Windows Phone. The Windows Phone version does not support synchronous requests, which are used in the Windows version, and it is not possible to simply assign query string data - it is instead necessary to build this into the query string.

  To solve these problems, a wrapper class, `SongHawk.WebData.WebClient`, was created using F# asynchronous workflows to convert asynchronous requests into synchronous method calls, and also maintaining a collection of query string parameters which are added to the request URL when the request is made. A preprocessor directive is then used to select which version of the `WebClient` class should be used when compiling the code for a specific platform.

- Unlike the various music libraries that are implemented for the Windows version, the music library that is available on Windows Phone 7 does not include years for songs in the library,

so it would not be possible to access this data without first finding it from some online data source. The decision was made to simply not use years at any point in the Windows Phone application, making the library connector throw an exception rather than returning data should that method be called.

The year attribute is used very rarely in the Windows version of the application, so it does not cause much functionality of the system to be missing.

- The MusicBrainz Sharp library cannot be compiled for the Windows Phone 7 platform, so again, the decision was made to simply not include MusicBrainz data in the Windows Phone application. In the Windows version of the application, MusicBrainz provides a very limited set of information, and often cannot find any of the requested information for a given artist or album. Most of the information that could have been provided by MusicBrainz is instead found from other sources.

5.4 Summary

In this chapter, the design and implementation of the application for Windows Phone 7 was discussed, along with the implementation details for accessing the library. The various issues that were discovered while converting the application to work on Windows Phone were also discussed, along with any workarounds that were taken.

Chapter 6

Testing

To ensure that the code written for this project does as is intended, testing was used throughout the project. Unit tests were used for the analytical functions and some library interactions, however these methods are not suitable for testing UI or web data due to the variability of the results. To test these parts of the system, black box testing was used instead.

6.1 Unit Testing

Unit testing was carried out on the system using the NUnit unit testing framework[28], with FsUnit[30] being used to write tests in a more functional manner. An example of a test can be seen in Program 15.

Unit tests were placed in a different assembly to separate them from the main code base, with a new assembly being created for each category of tests. The following are examples of unit tests that were created for the system:

- **Library Analytics:**
  - Check that the correct playcount statistics are returned for a set of dummy data.
  - Check that the correct song count statistics are returned for a set of dummy data.
  - Check that the correct duration statistics are returned for a set of dummy data.
  - Check that the correct year statistics are returned for a set of dummy data.
- **Library Plugins:**
  - Check library plugins are correctly composed using MEF.
  - Check that the data returned from library plugins is consistent with the contents of those libraries.
- **Library Selection Manager:**
  - Check that the various selection modifications leave the correct set of available items.
  - Check that the filtering functions return the correct subset of available items.
  - Check that all operations do not cause any unexpected side-effects.
- **Web Data:**
  - Check that XML parsing helper functions return the correct results.
  - Check that JSON parsing helper functions return the correct results.
- **Common:**
  - Check that utility functions give the correct results.

These tests were carried out regularly to ensure that changes to the code had not had any unintended side-effects and that no changes had occurred to prevent the correct results being returned.

Program 15 Testing of analytical functions using NUnit and FsUnit

```
namespace SongHawk.Test.Libraries

open SongHawk.Libraries.Library
open SongHawk.Libraries.Analytics
open SongHawk.Common.Levels
open NUnit.Framework
open FsUnit

[<TestFixture>]
[<Category("Library")>]
type ``Analytics Tests``() =

    [<TestFixtureSetUp>]
    let lib = Libraries() |> Seq.find (fun l -> l.GetName() = "Dummy Library")

    [<Test>]
    member test.``Playcounts should be retrieved for 2 genres``() =
        GetPlaycounts lib Level.Genre true [] [] []
        |> List.ofSeq
        |> should haveLength 2

    [<Test>]
    member test.``Genre 2 should have an average playcount of 4``() =
        let (_, v) =
            GetPlaycounts lib Level.Genre true [] [] []
            |> Seq.find (fun (k,_) -> k = ("Genre 2", null))

        v |> should equal 4

    [<Test>]
    member test.``Genre 2 should have a total playcount of 8``() =
        let (_, v) =
            GetPlaycounts lib Level.Genre false [] [] []
            |> Seq.find (fun (k,_) -> k = ("Genre 2",null))

        v |> should equal 8

    [<Test>]
    member test.``Artist 1 should have an average playcount of 4``() =
        let (_, v) =
            GetPlaycounts lib Level.Artist true [] [] []
            |> Seq.find (fun (k,_) -> k = ("Artist 1",null))

        v |> should equal 3.5

    [<Test>]
    member test.``Artist 1 should have a total playcount of 8``() =
        let (_, v) =
            GetPlaycounts lib Level.Artist false [] [] []
            |> Seq.find (fun (k,_) -> k = ("Artist 1",null))

        v |> should equal 7
```

6.2 Black Box Testing

The remainder of the system was tested using black box testing, checking that the various parts of the UI were displaying the correct data, and that the various UI interactions (button clicks etc.) did the correct things. This was done as an ongoing process, whenever changes were made to the code, various paths were taken through the system to ensure that the system was still behaving as expected. Particular focus was given to those areas of the code that had been changed.

The following are examples of black box testing that were performed during the course of this project:

- Check that the various UI controls perform the correct action when clicked/changed.
- Check that the data being returned from Web Data requests is consistent with the original data source (not suitable for unit testing due to the possibility that the data returned may have changed).
- Check that each screen is being populated with all of the expected data.

6.3 Summary

The testing that was completed during the course of this project helped to ensure that the code that was being written returned the correct results and could be put together to form a complete system without any unintended interactions. The use of unit testing and black box testing ensured that all parts of the code could be checked, even if they were not suitable for automated testing.

Chapter 7

Evaluation

In keeping with the dual aim of this project, the evaluation of the project is split into two areas: the applications as a whole (whether they meet the needs of the user etc.) and the language (how easy is it to use for this type of application etc.). These different evaluation areas are discussed in this chapter.

7.1 User Studies

To understand whether this project meets the needs of users and to identify areas where the applications could be improved or additional features could be added, a number of user studies were carried out. These involved users using the system, making use of the features that are available and seeing where the application is unintuitive or does not meet the user's needs.

After each user had used the system, they were asked the following questions:

- What features of the application did you particularly like?
- What features do you think the application is missing?
- Would you use the application?

The following questions were then asked for the individual platforms (Windows and Windows Phone):

- Are there any parts of the application that do not work as you would expect?
- Did you have difficulty using any parts of the application?
- Did you like the application layout?
- Did the navigation work as you expected?

These studies were carried out on three users and the results are described below.

7.1.1 What features of the application did you particularly like?

From the user studies, certain features were found to be liked more than others:

- The recommendations feature was found to be the most liked feature of the application as it was considered to be the most useful, allowing the user to find new artists that they might like.
- The rankings feature was considered to provide useful information about the user's listening habits.

- The ability to load the relevant section in Spotify was seen to be a good benefit, as the user can easily try any new music that they may have found through the other features.
- The release years statistics were seen to be a good feature, as it shows a simple view of how their music tastes range over a period of years.

7.1.2 What features do you think the application is missing?

Users suggested that there may be a few additional features that the application could benefit from having:

- The ability to see what parts of the library are least listened to, or least recently listened to. This would allow the user to rediscover parts of their library that they have not listened to much, or have not listened to in a long period of time.
- The ability to purchase music from Windows Phone, possibly through the Zune Marketplace. This would allow the user to expand their library based on recommendations.
- Integrate with media player control, giving the ability to play tracks from their library and also get quick information about the currently playing track.

7.1.3 Would you use the application?

In general, users found that the application has features that they would use, however opinions were split over whether they would be more likely to use the application on a computer or a mobile device.

One user only listens to music on their computer, not on a mobile device, so the application on a mobile device would not be able to provide them with any useful information due to the limited library size.

Two users felt that it is more difficult to look this information up on a mobile device, so an application to gather this information for them would be beneficial.

7.1.4 Are there any parts of the application that do not work as you would expect?

Windows

Generally, users found that the application worked as they expected, however there were a few small areas where functionality seemed to be incorrect, such as when filtering the library, as items were selected or deselected in one column, some items in the other columns deselect, apparently randomly.

It was also found that when using the iTunes plugin, any operation that required library access was slow, making the user dislike the amount of time being taken to render any results.

Windows Phone

One user felt that the behaviour of Panorama titles that fit on the screen is unusual when the screen is swiped across, as parts of the title disappear off the side. The advantage of this behaviour was however clearly seen when the title was longer.

The users also expected that the statistics pie charts would be ordered so that it is easy to see how items are ranked. This behaviour is currently inconsistent with the Windows version of the application.

7.1.5 Did you have difficulty using any parts of the application?

Windows

The users found that the application was generally easy to use, however at times it could be slow when accessing library data, particularly if lots of processing is necessary.

One user also felt that there could be some other way of modifying the library filtering rather than using checkboxes, perhaps something where multiple items can be selected at once.

Windows Phone

This version of the application was also generally found to be easy to use and performance was less of an issue, however it was commented that recommendations generally take too long to load. This is due to the amount of processing and web requests that are required to collect the necessary data.

7.1.6 Did you like the application layout?

Windows

In general, the layout of the application was well received, however there were a few points made:

- More space could be added around the information panes to separate it more from the information that is around the edges.
- It could be made "prettier", perhaps with the addition of more colour, however some users liked the monochrome style used.
- The combo box for selecting between statistics types should be more obvious. It can be difficult to notice that the option is there.
- The layout of the main screen was not liked by some users, with suggestions that it might be better if the options were arranged in a column.

Windows Phone

The layout of the mobile version of the application was also well received, but there were a few points made:

- The amount of screen used for displaying genre, artist and album descriptions could be increased to maximise the amount that could fit on the screen. There is currently a reasonably sized margin on the right, allowing the start of the next panel to be seen.
- Make the titles stand out more - they are too similar to pivot titles to be clear what the title is, and what the section headings are.
- The selection for the type of statistics (genre, artist, album) looks out of place as radio buttons. Use of a pivot control to switch between the types would be more consistent with the rest of the UI.
- The legend title could be removed from the statistics pie charts - there is no benefit to it being there, and it would allow the chart to be slightly bigger.

7.1.7 Did the navigation work as you expected?

Windows

The navigation in the system was found to generally be good, with features being easily discoverable, however there were some points about some specific parts of the navigation:

- It could be possible to change between the different types of statistics without having to go back to the main statistics screen.
- The pie chart segments should be able to be clicked to access information pages, like in the Windows Phone version.
- The style of a button should change when it has been clicked so that the user can be certain that they have clicked in the right place.

Windows Phone

Navigation on the Windows Phone version was considered to be good, however one user found that it was sometimes difficult to know what would be displayed if they were to swipe across, loading another horizontal segment of a screen.

7.1.8 General Comments

As well as the answers given to the above questions, the users were also given the opportunity to make general comments about the application:

- Genres could be featured less prominently, as it is often the case that music libraries contain music that has been incorrectly assigned a genre, or even that certain albums or tracks span multiple genres.
- When no information is found, a "No information found" message is shown. This could be made more specific, e.g. "Last.fm has no information on this topic".
- It would be useful to have some form of feature overview for new users. This could be combined with the main page, describing each feature rather than just giving a button to access it.
- When displaying data in pie charts, it may be more useful to only show a certain number of items, grouping the rest together, as it is currently the case that some items are far to small to appear in any helpful manner.
- It would be good if was available on additional platforms, such as Linux or Android.

7.1.9 Summary

Both versions of the application were generally well received, with users saying that the application has features that they would like to use. A number of comments were raised by the users, mainly minor things that would be relatively easy to change and the users also provided a set of additional features that could be added to the application to make it more useful.

7.2 The Language

To evaluate the suitability of F# for a project of this type, the following areas will be looked at:

- How difficult is it to use F# to creating complex systems with UI and data processing?

- What differences are there between using F# to develop for Windows and Windows Phone?
- Does F# allow a reduction in the amount of code written compared to C#?
- How do the features of F# compare to other languages, such as Java and Haskell?

7.2.1 How difficult is it to use F# for this type of application?

Throughout the project, it has been found that F# is generally a very capable language that can be used in many different ways to create complex systems such as this, with a user interface and processing of the underlying datasets.

There was very little difficulty involved in using F# for this project, with it being used for the whole system, apart from a small wrapper for deploying the Windows Phone version. A wide range of features of the language were used throughout the project, with easy interoperability with the various external dependencies and third party libraries.

No issues revolving around the use of the language were identified at any point during the project, largely due to the multi-paradigm approach that can be taken when writing F# code. If it is needed to store state, or pass around and create objects, this is all possible with F#, despite it being considered a primarily functional language. The functional features of the language did play a large part in the development of the applications, with much of the core data processing involving list processing, which can be done very simply and without side-effects, and pattern matching was used extensively. These types of programming can be much more complex when writing non-functional code, for instance, although switch statements can perform the same actions as some simple forms of pattern matching, it is not possible to use different types of comparison - it always checks equality.

The ability of F# to interoperate with other .NET libraries is a great strength of the language, as this allows F# programs to take full advantage of the large number of .NET libraries (both built-in and third-party) that are available. Although F# introduces new data types and syntax for declaring them, they usually conform to standard .NET types, often incorporating generics, and as such can often be passed into other code with ease, for example F# lists, declared using the `[]` syntax implement the generic .NET list type, so can be used anywhere that a list is expected.

This ability to interoperate also allows the wide range of frameworks that are part of the .NET Framework to be used, including Windows Presentation Foundation and the Managed Extensibility Framework. These are used to provide much of the functionality of the application, and do not require any special techniques to be used from F# code.

The code to create the user interface for the Windows version of the application also shows the flexibility of some of these frameworks. Although Windows Presentation was designed to work with an object model whereby functionality is added to various controls by implementing classes of these types, the approach taken with this project generates the objects from a XAML file, and then adds the functionality on after finding these objects in the tree. It is possible to add all of the functionality that is usually added by writing object-oriented code in a much more functional style, although these methods do have side-effects, so cannot truly be functional.

F# allows the developer to take full advantage of the many libraries that are available and has many language features that make it easier to perform specific tasks related to application development. The use of F# for this type of project does not have any noticeable disadvantages over using other .NET languages, such as C#, due to having the same underlying libraries. If anything, the additional features available in the language make it easier to perform certain tasks in the development of such a system.

7.2.2 What differences are there between using F# to develop for Windows and Windows Phone?

Although Windows Phone is a distinct platform from the full .NET Framework used for the Windows application, the two systems share many libraries and can be used in very similar ways.

The main differences encountered between the two platforms are:

- **Slightly different methods on some classes:** Although many of the classes that are available in the libraries provided for Windows Phone correspond to classes provided in the .NET Framework, some of these classes have slight differences, for example the `WebClient` class only has asynchronous methods on Windows Phone, but has synchronous methods on Windows.
- **Different core libraries:** As the Windows Phone core libraries are a subset of the .NET Framework's core libraries, it is not possible to directly take .NET assemblies and use them for Windows Phone. It is therefore necessary to compile these libraries specifically for Windows Phone. Although this is easily done with the assemblies created for this project, it is not always possible to simply recompile a third party library to work on the platform, for example, the MusicBrainz Sharp library uses some core methods of the .NET Framework that are not available on Windows Phone.

As well as differences between the two platforms, there are also a number of similarities:

- **Most code compiles for both platforms:** The vast majority of the code written for the project could simply be compiled for each platform without requiring any changes. Incompatibilities between the two could usually be solved quite simply, either by changing the use of a method, or by creating a wrapper to the Windows Phone class to make it compatible with the methods used from the .NET Framework version.
- **Similar UI Frameworks:** The user interface framework used on Windows Phone (a derivative of Silverlight) has a very similar style to that used by the Windows Presentation Foundation for Windows. Most of the controls that are available for WPF are also available for Windows Phone and similar XAML can be written for each platform. The differences between the two are largely to deal with the different types of device, so the same techniques can usually be used for creating the UI and populating the data.

These similarities mean that it is fairly simple to take a Windows application developed in F# and port it to Windows Phone. Code often just works and the libraries available are similar on both platforms meaning that similar techniques can be used.

7.2.3 Is F# more concise?

To evaluate whether or not code can be written in a more concise manner using F#, this section compares sections of the project code that are written in F# with the equivalent code written in C#. Various areas of the code base are compared to see whether there are some tasks that are better implemented in F#, or for which F# is not the best choice of language. Generic examples of common tasks are covered first, before moving on to specific examples from the project code.

Filtering Lists

Filtering lists is an technique that is used throughout the code to reduce data that is being processed down to a smaller set of data that we are interested in. In F#, this can easily be done using the `List.filter` function, with either a function reference or an anonymous function being passed in to perform the filtering (see Program 16). This cannot be done so easily in C#, with it being necessary to iterate through the list, adding any relevant items into a new list, as shown in Program 17. Overall, this leads to a decrease in code size when using F#, as there is much less syntax required to achieve the same thing.

Pattern Matching

Pattern matching is also used throughout the code, often to select a specific piece of data from an object based on some criteria. F# pattern matching allows a variable to be compared against any

Program 16 F# list filtering

```
// For listOfData <- [1,2,3,4,5], we should get [4,5]
listOfData |> List.filter (fun item -> item > 3)
```

Program 17 C# list filtering

```
// For listOfData <- [1,2,3,4,5], output should equal [4,5]
List<int> output = new List<int>();
foreach(int item in listOfData) {
    if(item > 3) {
        listOfData.Add(item);
    }
}
```

supported pattern, such as checking for being equal to some other item, or for extracting data from a list or other data structure. This cannot be expressed so simply in C#, with `switch` statements being the closest comparable feature. The `switch` statement does have limitations, with it only being able to deal with equality of items, and only working on certain types such as booleans or strings. Program 18 shows a function that uses pattern matching to process data from a list, and Program 19 shows the C# version. The F# version also uses the `function` syntax to avoid the need to create a parameter and use `match paramName with`.

Program 18 Processing a list with F# pattern matching

```
let ProcessList = function
    | [] -> "Empty List"
    | i :: [] -> sprintf "Single item: %s" i
    | i :: _  -> sprintf "Multiple items, first: %s" i
```

Rankings

The rankings code is one area of the code base where a comparison between F# and C# shows clear results. The F# version of this can be seen in Program 20 and the C# version can be seen in Program 21 (only a subset of this code is included). Due to the inability to use the higher order list functions that are available in F#, the C# version uses more loops, usually adding new objects to lists. This requires much more boilerplate code for maintaining lists of data and constructing new objects. The clear benefits of pattern matching can be seen in the `Query` function, where it is necessary to instead use nested if statements in the C# function due to the limitations of the `switch` statement.

Saving Settings

For some areas of the code, the C# style of programming is much more suited to the task than the more functional F# style. This does not present an issue as where necessary C# code can easily be translated to F# code, with the same structure and the same function calls. This technique was used in parts of the UI code, particularly when handling state. For example, when saving settings from the Options window, the code involves a lot of getting data from the UI and writing it back to the settings storage (the F# code is shown in Program 22 and the C# code in Program 23). Here, the only real difference between the two is to do with the syntax, with C# having additional braces and semicolons. The code that makes the window functional has the same structure.

Summary

In general, when interacting with libraries that were not created with F# in mind, such as user interfaces, using F# does not lead to a smaller amount of code, however when dealing with data

Program 19 Processing a list with C#

```
string ProcessList(List<string> list) {
    if(list.Count == 0) {
        return "Empty List";
    }
    else if(list.Count == 1) {
        return "Single item: " + list[0];
    }
    else {
        return "Multiple items, first: " + list[0];
    }
}
```

Program 20 Rankings code in F#

```
let private NormalisePairSeq (list:PairSeq) =
    let mean = list |> Seq.averageBy GetValue
    let standardDeviation = StandardDeviation list mean

    list |> Seq.map (fun (k, v) -> (k, (v-mean)/standardDeviation))

let private Query (lib:ILibrary) level genres artists albums length =

    match lib.GetTracks genres artists albums with
    | SeqEmpty -> []
    | tracks   ->
        let FindKey =
            match level with
            | Level.Genre  -> fun (t:ITrack) -> Single t.Genre
            | Level.Artist -> fun (t:ITrack) -> Single t.Artist
            | Level.Album  -> fun (t:ITrack) -> Pair (t.Artist, t.Album)
            | _            -> fun _          -> None

        let (songcounts, durations, playcounts) =
            tracks
            |> Seq.map (fun t ->
                (FindKey t), float (t.Length), t.Playcount
            )
            |> Seq.groupBy (fun (k, _, _) -> k)
            |> Seq.fold
                (fun (cs, ds, ps) (k, vs) ->
                    cs |> Seq.append [(k,double (vs |> Seq.length))],
                    ds |> Seq.append [(k,double (vs |> Seq.sumBy (fun (_, d, _) -> d)))],
                    ps |> Seq.append [(k,double (vs |> Seq.sumBy (fun (_, _, p) -> p)))]
                )
                (Seq.empty, Seq.empty, Seq.empty)

        Normalise songcounts durations playcounts
        |> ApplyWeightings
        |> SumResults
        |> Seq.sortBy (fun (_,v) -> -v)
        |> tryTake length
        |> Seq.toList
```

Program 21 Rankings code in C#

```
private static PairSeq NormalisePairSeq(PairSeq list) {
  double total = 0;
  foreach (Tuple<Rankings.RankingsKey, double> item in list) {
    total += item.Item2;
  }
  double mean = total / list.Count;
  double standardDeviation = StandardDeviation(list, mean);
  PairSeq output = new PairSeq();
  foreach (Tuple<Rankings.RankingsKey, double> item in list) {
    output.Add(new Tuple<Rankings.RankingsKey, double>(item.Item1,
        (item.Item2 - mean) / standardDeviation));
  }
  return output;
}

private static PairSeq Query(ILibrary lib, Levels.Level level, IEnumerable<String> genres,
      IEnumerable<String> artists, IEnumerable<String> albums, int length) {
  List<ITrack> tracks = (List<ITrack>)lib.GetTracks(genres, artists, albums);
  if (tracks.Count == 0) {
    return new PairSeq();
  } else {
    PairSeq songcounts = new PairSeq();
    PairSeq playcounts = new PairSeq();
    PairSeq durations = new PairSeq();
    foreach (ITrack t in tracks) {
      Rankings.RankingsKey key;
      if (level == Levels.Level.Genre) {
        key = Rankings.RankingsKey.NewSingle(t.Genre);
      } else if (level == Levels.Level.Artist) {
        key = Rankings.RankingsKey.NewSingle(t.Artist);
      } else if (level == Levels.Level.Album) {
        key = Rankings.RankingsKey.NewPair(new Tuple<string,string>(t.Artist, t.Album));
      } else {
        key = Rankings.RankingsKey.None;
      }
      int index = songcounts.FindIndex((Tuple<Rankings.RankingsKey, double> value) =>
          value.Item1 == key);
      if (index == null) {
        songcounts.Add(new Tuple<Rankings.RankingsKey, double>(key, 1));
        durations.Add(new Tuple<Rankings.RankingsKey, double>(key, t.Length));
        playcounts.Add(new Tuple<Rankings.RankingsKey, double>(key, t.Playcount));
      } else {
        songcounts[index] = new Tuple<Rankings.RankingsKey, double>
            (songcounts[index].Item1, songcounts[index].Item2 + 1);
        durations[index] = new Tuple<Rankings.RankingsKey, double>
            (durations[index].Item1, durations[index].Item2 + t.Length);
        playcounts[index] = new Tuple<Rankings.RankingsKey, double>
            (playcounts[index].Item1, playcounts[index].Item2 + t.Playcount);
      }
    }
    var normalised = Normalise(songcounts, durations, playcounts);
    var weighted = ApplyWeightings(normalised.Item1, normalised.Item2, normalised.Item3);
    var summed = SumResults(weighted.Item1, weighted.Item2, weighted.Item3);
    summed.Sort();
    summed.Reverse();
    return (PairSeq)summed.Take(length);
  }
}
```

Program 22 Saving settings in F#

```
let private SaveClicked (w:Window) e =
    let panel = FindControl "LibrarySources" w :?> StackPanel
    let defaultLib = FindControl "ComboDefaultLibrary" w :?> ComboBox
    let useSpotifyHttp = FindControl "CheckUseSpotifyHttp" w :?> CheckBox

    let disabledLibs =
        [for i in 0..panel.Children.Count-1 do
            let c = panel.Children.Item i :?> CheckBox
            if not c.IsChecked.Value then
                yield c.Tag.ToString()
        ] |> List.toArray

    if disabledLibs |> Array.length = panel.Children.Count then
        MessageBox.Show(
            w,
            "You must enable at least one library plugin!",
            "Error",
            MessageBoxButton.OK,
            MessageBoxImage.Error
        ) |> ignore
    else
        let settings = ApplicationSettings()
        settings.DisabledLibraries <- disabledLibs
        settings.DefaultLibrary <- defaultLib.SelectedItem :?> string
        settings.UseSpotifyUris <- not (useSpotifyHttp.IsChecked.GetValueOrDefault())
        settings.Save()
        w.DialogResult <- nullable true
        w.Close()
```

Program 23 Saving settings in C#

```
private void SaveClicked(Window window, RoutedEventArgs e) {
  StackPanel panel = (StackPanel)window.FindName("LibrarySources");
  ComboBox defaultLib = (ComboBox)window.FindName("ComboDefaultLibrary");
  CheckBox useSpotifyHttp = (CheckBox)window.FindName("CheckUseSpotifyHttp");

  List<string> disabledLibs = new List<string>();
  foreach (var i in panel.Children)
  {
    CheckBox item = (CheckBox)i;
    if (!item.IsChecked.GetValueOrDefault()) {
      disabledLibs.Add(item.Tag.ToString());
    }
  }

  if (disabledLibs.Count == panel.Children.Count)
  {
    MessageBox.Show(
      window,
      "You must enable at least one library plugin",
      "Error",
      MessageBoxButton.OK,
      MessageBoxImage.Error
    );
  }
  else
  {
    ApplicationSettings settings = new ApplicationSettings();
    settings.DisabledLibraries = disabledLibs.ToArray();
    settings.DefaultLibrary = defaultLib.SelectedItem.ToString();
    settings.UseSpotifyUris = !useSpotifyHttp.IsChecked.GetValueOrDefault();
    settings.Save();
    window.DialogResult = true;
    window.Close();
  }
}
```

processing, or with libraries that were designed for use with F#, a reduction in code size can be achieved as the various specific features of F# allow these areas to be written in a very concise manner. F# code does however generally have less syntactic sugar, as for example, braces are uncommon in the language (being used for marking asynchronous code blocks), with white space being used to handle things such as scoping.

7.2.4 How does F# compare to other languages?

There are many different ways in which F# can be compared with other languages: what features are unique to F#, what is missing from F# and what features do not provide any advantage (or disadvantage) against other languages? Some of these comparisons are described below.

COM Interoperability

As F# runs on the .NET Framework, it can use APIs that are provided for accessing data from libraries through COM without having to rely on any external libraries to call down to the underlying Windows COM API. In other languages that cannot directly access this API, this is less straightforward, although libraries have been created for some languages, such as com4j[33] or JACOB[34] for Java.

These libraries do however make it fairly simple to use COM APIs, generating code to allow the various methods to be accessed. These libraries provide code generators to create an intermediate layer which allows access as shown in Program 24. To do this in F#, the COM API can simply be added as a project reference, and the methods immediately become available without the need to use code generators. The code in Program 25 does the same as the com4j example.

These libraries can also be used by other languages that run on the Java Virtual Machine, such as Scala and Jython through the methods that these languages use to call Java code.

Program 24 Accessing the iTunes COM API from Java using com4j[33]

```
import iTunes.def.ClassFactory;
import iTunes.def.IITTrack;
import iTunes.def.IiTunes;

public class Main {
    public static void main(String[] args) throws Exception {
        IiTunes iTunes = ClassFactory.createiTunesApp();

        IITTrack track = iTunes.currentTrack();
        if(track==null) {
            System.out.println("Nothing is playing");
        } else {
            System.out.println("Now playing: "+ track.name());
        }
    }
}
```

Although F# has built in support for COM interoperability, other languages can be provided with this support through the use of third party libraries, and as such, the use of F# does not provide any real advantage to using other languages.

Multi-Paradigm Programming

Although this project makes heavy use of the functional features of F#, such as immutability of variables and most of the core functionality uses functions that do not have side-effects, it is also the case that the user interface code makes use of some of the object-oriented and imperative

Program 25 Accessing the iTunes COM API from F#

```
open iTunesLib

let iTunes = new iTunesAppClass()
let track = iTunes.CurrentTrack
if track = null then
    printfn "Nothing is playing"
else
    printfn "Now playing: %s" track.Name
```

programming styles: mutable variables being used to store the current system state, and objects being used as the basis of the Windows Phone 7 user interface implementation. This is one of the key advantages to F# as a language: fully functional code can be written, but often you need to be able to have a bit more flexibility, particularly for interacting with other languages and libraries.

The functional aspects of this project could easily have been written in another functional language, Haskell, however this would have meant that it would have required other parts of the code to be written in a different language if they were to be written in a simple manner. Storing global mutable state can be done through the use of `unsafePerformIO`, however this is far from an ideal way of doing this, with various other techniques used for simulating this behaviour[4].

Although object-oriented code did not necessarily need to be written for this project, the use of objects did help in certain places. The use of objects for the `LibrarySelectionManager` meant that it was not necessary to store a number of mutable variables, instead storing the state within an object that could be passed around without any issues. It also made the use of built in functions easier, for instance, the use of objects for Windows Phone pages meant that the built-in navigation system could be used to keep track of the user's actions. Without this, it would have been necessary to create additional code to keep track of actions, which would be likely to have mutable variables in it, hence violating one of the core features of functional programming.

This multi-paradigm approach is very similar to OCaml, and as such, much of the code of this project could be translated easily. The main advantage of F# over OCaml in this case is the ability of F# to consume .NET libraries, which are used to provide much of the functionality of the application.

Asynchronous Workflows

The ability to use asynchronous workflows in F# makes asynchronous programming far easier than in the standard models used by Java or C#. In most languages, it is necessary to use callback functions that will be executed once an asynchronous method call has been completed, however with the F# `async` keyword and associated functions, callbacks are no longer necessary as execution will continue once an asynchronous block has completed.

This asynchronous approach is used throughout the project to allow execution to occur in background threads, without blocking the UI thread. F# provides an `Async.SwitchToContext` function to easily allow execution from background threads to switch to the UI thread for updating the UI, and it is very simple to cause code to be executed in other threads, there is no need to use callback functions at any point.

A similar approach is being added to C# and Visual Basic.NET, by marking methods as being asynchronous and using the `await` keyword to wait for the response. As such this easy method of dealing with asynchronous workflows is becoming available in other languages.

Summary

It is clear to see that there are some features, such as asynchronous workflows, that allow F# to stand apart from other languages, and there are benefits to using F# over other functional or multi-paradigm languages. It is also the case that many of the things that can be done with F#

can be done just as easily with other languages, and indeed, other languages may better support some things. F# is a good all-round language that is certainly as capable as other languages, and has many benefits that come with running on the .NET Framework.

Chapter 8

Possible Future Work

There are a number of areas in which further work could be done on this project, including:

- **Least Listened To Music:** In addition to the current top rankings feature, the application could have a feature to highlight the least listened to tracks, or least recently listened to tracks, allowing the user to rediscover parts of their library that they do not listen to often.
- **Player Control:** The application could be extended to control the tracks that are currently playing in the media player, and allow the user to easily view information about the currently playing track or album.
- **Social Networking:** The application could be integrated into social networks, allowing users to share suggestions of artists or albums with other people by, for example, sending a tweet or using the Last.fm shout feature.
- **Additional Platforms:** The application could be ported to additional platforms, for example, Linux or OSX using Mono, or onto other mobile devices through the use of tools such as MonoTouch or Mono for Android.
- **Centralised Data Processing:** Some of the data processing that is currently done in the application could be moved to a central server where better caching can be done of data to allow information to be generated more quickly and reduce the amount of HTTP traffic necessary. This has particular advantages for the recommendations feature, as a list of artists could be sent to the server, and a single response would provide the results rather than having to make multiple HTTP requests before being able to calculate results.
- **F# Type Providers:** Converting the code to use type providers would reduce the complexity of some of the data acquisition code, reducing the amount of code and increasing the reliability.

Looking into these areas would allow the application to become more useful, with added functionality, and also increase the speed at which parts of the application work.

Chapter 9

Conclusion

This project had two aims: to develop a piece of software that can provide additional information about music library contents and to evaluate the F# language to see how difficult it is to use for writing real-world pieces of software.

To meet these aims, two different applications were developed, with a large amount of common code - one running on Windows under the .NET Framework, and the other running on Windows Phone 7. These applications had a very similar feature set and much of the analysis code was able to be compiled to run on both platforms. They were entirely written in F#, making an evaluation of the suitability of the language cover many different language features and showing that it is not necessary to switch to other languages during development.

For the first aim, the applications developed were evaluated through the use of user studies and were found to provide a feature set that was liked by the users. Most aspects of the applications were liked by the users, however a number of comments and suggestions were made, including additional features that could be added and parts of the system that do not work quite as well as they could.

From the user studies conducted, a number of different areas in which the application could be extended have been identified: providing details of less commonly listened to music, integration with media player controls and porting to additional platforms. The slowness of certain parts of the application could also be addressed, possibly through the use of a central server for processing some of the more complicated parts (such as recommendations) as some details could be cached for all users, reducing the load on the APIs, and reducing the number of requests needed to be made by the application.

For the evaluation of the language, F# was found to be very capable for this type of task, with a number of language features to make parts of the code very easy to write. It was able to be used for the whole system, largely due to it's multi-paradigm approach which meant that it could integrate with third party libraries and be written in a manner that would require a different language if a fully functional language was being used.

In general, F# was quite easy to learn (although knowledge of a language such as Haskell and experience with the .NET libraries probably helps) and it was generally a nice language to write in. It is easy to perform most types of tasks, be they fully functional or more object-oriented, allowing functionality to be quickly and easily added to applications.

This project can be considered to be a success - both of the aims were completed, with a good result in both, with applications that are liked by the user and an evaluation of the language showing that it can easily be used for real-world applications and is not limited to fields such as data processing. Completing this project has been fulfilling experience, giving experience in another language and using features of both languages and platforms to produce applications that can be used by real users to try and improve their listening experiences.

Bibliography

[1] Silverlight Toolkit. http://silverlight.codeplex.com/. Additional controls for Silverlight (including Windows Phone), including charts.

[2] WPF Toolkit. http://wpf.codeplex.com/ [Accessed 6 January 2011]. Additional controls for WPF applications, including charts.

[3] Metadata Backup Open Source Project. http://sourceforge.net/projects/metadatabackup/ [Accessed 17 November 2010], April 2010. Example of accessing the Windows Media Player library using C#.

[4] Adrian Hey, Brett Giles, Ashley Y et al. Top Level Mutable State. http://www.haskell.org/haskellwiki/Top_level_mutable_state [Accessed 10 June 2011]. A discussion of storing mutable state in Haskell.

[5] MusicBrainz Community. MusicBrainz Sharp. http://musicbrainz.org/doc/Musicbrainz_Sharp [Accessed 3 January 2011], April 2009. .NET library for accessing MusicBrainz data.

[6] MusicBrainz Community. XML Web Service. http://musicbrainz.org/doc/XML_Web_Service [Accessed 3 January 2011], September 2010. MusicBrainz XML Web Service documentation.

[7] Microsoft Corp. *UI Design and Interaction Guide for Windows Phone 7*. 2010. Details of the "Metro" user interface for Windows Phone 7.

[8] Dan Crevier. Automating/scripting iTunes on Windows. http://blogs.msdn.com/b/dancre/archive/2004/05/08/128645.aspx [Accessed 17 November 2010], May 2004. C# example of using the iTunes COM library.

[9] Don Syme, Adam Granicz, Antonio Cisternino. *Expert F# 2.0*. Apress, 2010. Book covering all of the main aspects of programming in F#.

[10] Don Syme, Gregory Neverov, James Margetson. Extensible Pattern Matching via a Lightweight Language Extension. In *International Conference on Functional Programming*, 2007. Paper describing the active patterns feature of the F# language.

[11] Don Syme, Tomas Petricek, Dmitry Lomov. The F# Asynchronous Programming Model (Draft). In *Practical Aspects of Declarative Languages*, 2011. Paper describing the asynchronous programming model in F#.

[12] Natty Gur. Pluggable Architecture. http://weblogs.asp.net/ngur/archive/2004/08/02/205789.aspx [Accessed 14 January 2011], August 2004. Overview of pluggable architectures.

[13] Lars Hildebrandt. WpfSimpleChart Library. http://wpfsimplechart.codeplex.com/ [Accessed 6 January 2011]. WPF chart controls library.

[14] Apple Inc. iTunes COM for Windows SDK. http://developer.apple.com/sdk/itunescomsdk.html [Accessed 17 November 2010], September 2009. Documentation and code samples for using the iTunes COM interface.

[15] Google Inc. Freebase API. http://www.freebase.com/docs/mql/ch01.html [Accessed 3 January 2011], 2010.

[16] Google Inc. Freebase Music. http://www.freebase.com/view/music [Accessed 3 January 2011], 2011. Music section of the Freebase website.

[17] Jack Gudenkauf, Jesse Kaplan. .NET Application Extensibility. *MSDN Magazine*, February 2007. Introduction to extensibility using System.Addin.

[18] James Moore. F# running on the iPhone. http://blog.restphone.com/2009/09/f-running-on-iphone.html [Accessed 30 May 2011], September 2009. Example of F# code running on iOS through MonoTouch.

[19] James Newton-King. Json.NET. http://json.codeplex.com/.

[20] Jomo Fisher. Neat Samples: Extend your F# program with MEF. http://blogs.msdn.com/b/jomo_fisher/archive/2010/03/09/neat-samples-extend-your-f-program-with-mef.aspx [Accessed 14 January 2011], March 2010. Sample of using MEF with F#.

[21] Last.fm Ltd. Last.fm Web Services Overview. http://www.last.fm/api/intro [Accessed 3 January 2011], February 2009. Last.fm Web Service documentation.

[22] Spotify Ltd. Metadata API. http://developer.spotify.com/en/metadata-api/overview/ [Accessed 3 January 2011]. Documentation for Spotify's Metadata API.

[23] Microsoft Corp. Managed Extensibility Framework Overview. http://mef.codeplex.com/wikipage?title=Overview [Accessed 14 January 2011], December 2009. Overview of MEF.

[24] Mono Project/Novell. Gui Toolkits. http://www.mono-project.com/Gui_Toolkits [Accessed 30 May 2011]. List of GUI toolkits available for use on Mono.

[25] Mono Project/Novell. Mono. http://www.mono-project.com/Main_Page [Accessed 30 May 2011]. Mono project website.

[26] Mono Project/Novell. WPF. http://www.mono-project.com/WPF [Accessed 30 May 2011]. Current status of Mono plans for WPF.

[27] Microsoft Developer Network. Windows Media Player SDK. http://msdn.microsoft.com/en-us/library/dd758070.aspx [Accessed 17 November 2010], September 2010. MSDN Windows Media Player SDK documentation.

[28] NUnit Development Team. NUnit. http://www.nunit.org/ [Accessed 12 Jauary 2011]. NUnit .NET unit testing framework.

[29] Robert Pickering. *Foundations of F#*. Apress, 2007. Book covering all of the main aspects of programming in F#.

[30] Raymond Vernagus. FsUnit. http://fsunit.codeplex.com/ [Accessed 12 January 2011]. Add F# style syntax to NUnit unit tests.

[31] Don Syme. The Future of F#. In *F# in Education*. Microsoft, November 2010. Talk on the Type Providers which are being added into the language. http://research.microsoft.com/apps/video/default.aspx?id=141408 [Accessed 23 November 2010].

[32] Cameron Taggart. iTunes via F# Interactive. http://blog.ctaggart.com/2010/08/itunes-via-f-interactive.html [Accessed 17 November 2010], August 2010. Example code for accessing iTunes library data from F#.

[33] The com4j Project. com4j. http://com4j.java.net/ [Accessed 10 June 2011]. A COM library for Java.

[34] The JACOB Project. JACOB - Java COM Bridge. http://sourceforge.net/projects/jacob-project/ [Accessed 10 June 2011]. A COM library for Java.