

Software Maintenance Manual

Contents

- 1 Setting up the software Development environment
- 2 Setting Up the Project
- 3 Running the project
- 4 File descriptions
- 5 Known bugs

1. Setting up the software development environment

There are 3 pieces of software that you need to be being able to edit and compile the project in a development environment.

1. Microsoft Windows software development kit
2. Microsoft DirectX software development kit
3. Microsoft C# 2005 Express edition

All of these are free to download from the Microsoft website. For convenience I have submitted these CD's Containing the software with this report. If the windows development kit will not install because the .net framework is not installed on your computer, then the .net framework can be found on the runtime CD that was submitted with the project. Alternatively use the windows update feature to install the .net framework, version 3.0

I would suggest installing the windows SDK first, followed by the DirectX SDK and finally C# Express Edition. Use all the default settings during the installation where appropriate

2. Setting up the project

Step 1

Extract the honoursproject folder from the submitted code archive `meisel_david.tar.gz`.

Step 2

Start the c# .NET 2005 development environment. The development environment might ask some questions the first time it is run. Where appropriate just use the default options

Step 3

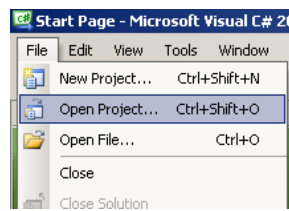


Figure 1 Open Project

When the c#.net application has loaded up, add the project to the development environment. This is done using the open project command which is located under the file menu as shown in figure 1. Browse to the honoursproject folder that you extracted in step 1 and locate the project.sln file. Open this file and the relevant project files will automatically be loaded into the development environment.

3. Running the project.



Figure 2 Start the project

To run the jukebox within the development environment simply press the green play button on the tool bar. This will automatically compile the project before running it. Alternatively under the debug menu you can select either start with debugging, or start without debugging.

The main class within the jukebox application is jukebox.cs. This is the file that the development environment starts with when the green play button is pressed.

Please refer to the user manual for instructions on how to use the application. The DirectX development kit automatically installs the DirectX assemblies described in the user manual

4. File descriptions

About.cs

This is a simple dialogue box with an authorship statement in it.

Album.cs

The album.cs class is required by the amazonitem.cs class. This class should be populated first. This is done by passing an instance of xml reader, containing xml from an Amazon request about an album. This class provides various methods available to the music recommender to allow it to determine information about the contents of the album. This class sits in between the Amazon

music recommender and the disc class therefore it does not directly hold any information about the tracks on the disk.

Amazon.cs

The amazon.cs class is the main interface between the Amazon web services and the jukebox. This class allows calls to be made to the Amazon web services by taking special search and response criteria. When a request is made, a Uri string is generated, a http request is then made with the uri and the response is read into a stream. This stream is then parsed in the appropriate method for the request. The amazon.cs class is heavily dependant on other classes for parsing special items of data. The purpose of this is to allow easy expandability for information that we might want in the future, and discardability for data that we are just not interested in.

amazonEngine.cs

The amazonEngine class coordinates the recommendation and contains most of the logic for the music recommender. The class works generally by making requests to Amazon for information about tracks on CD's and what CD's others bought based on a given CD. The class will filter this information down in each step discarding tracks and albums that are of no interest to us. This allows us to determine recommendations by cross-referencing music that is loaded into the jukebox.

amazonItem.cs

The amazonitem.cs class is called directly by the amazon.cs class. The purpose of this class is to take an xml response from the Amazon web services and populate all the relevant data structures. The class will take an instance of the xml reader, each method that is called to process the relevant part of the data structure will take this xml reader, the method will take off the relevant data that it needs to populate its own parts of the data structure. A given method, will the wind the xml reader forward so that the xml reader is in a state for the next method to extract data that it needs without having to search for it.

amazonPreferences.cs

This class is a dialogue box that allows the user to change various parameters for the Amazon music recommender, New values are chosen via a scroll bar and are made final when the user clicks ok. The class automatically updates the values in the textbox when the relevant scrollbar is changed.

AmazonSimilarity.cs

The purpose of this class is to take an xml wound forward to an item that has been recommended as part of an Amazon request. This class will then process this item, in its current state all information is discarded except for the ASIN which allows us to make a more in detailed request to amazon in subsequent requests. You need an instance of this class for every similarity that is recommended to you.

awCriteria.cs

The purpose of this class is to allow the programmer to dynamically create a search criteria string rather than having one coded. Calls should be made to the various set methods with given criteria.

There is no order in which these calls need to be made. Once all calls have been made the programmer should request that they wish the criteria string. The class will generate a criteria string based upon the set methods that the user has called and the corresponding data that the set methods were called with.

awResponseGroup

In a similar way to the awCriteria class this one allows the programmer to dynamically create the response group string. Calls should be made to the various set methods, however there is no need to pass any parameters. There is no need to make calls to the set methods in any given order. Once all the desired sets have been called the getString method should be called. This takes care of generating the response group part of the request.

Byteoperations.cs

This method provides an easy way to get subsections of byte arrays and to convert byte arrays into strings

Disk.cs

The purpose of the disk class is to conceptually represent a physical disk within the program. The caller usually constructs the class with album name or album and artist name, the users can then add tracks to this class. Please note that tracks must be added in the same order that they are on the original disk, if this causes problems all tracks can be cleared from the album and you can start repopulating the data structure. By default an instance is focused on the first track in the disk. This means that any calls made about a given track on the disk will be specific to that first track. To change the focus to subsequent tracks you should first test to see if there are subsequent tracks, if there are then you can call the iterator to change the focus of the instance to the next track. If you wish to go back to a previous track then you must reset the iterator, then advance through the tracks until you find the one you are looking for. This class needs to be programmed in this round about way because there is no way to determine in advance how many tracks there will be on the disk.

Fileloader.cs

This class provides various functions to the gui to enable users to add files. In the case that the user wants to simply add one file then, a request is made to have this file added to the jukebox. In the case that the user want to add a whole folder then this class must work out all files in the folder. If a recursive add is selected then all child folders must be searched for files also. We then request that this list of files is added to the jukebox library. This class also contains trivial functions to assist certain tasks in the class

Files.cs

This class was created as a central reference for all files in the jukebox. Leaving static references throughout the code was becoming troublesome when the application was being moved between folders and computers. The advantages of having all the file references in one location is that if one needs to be changed, It can be changed once and the change permeate throughout the application.

FileValidation.cs

This class is a trick that will exclude non mp3 files from the library based on extension. The class is not fool proof, it simply adds efficiency to the file add process by simply rejecting files that do not have an .mp3 extension. If anyone wanted to modify the program to play other types of files then the changes should be made in this class.

Google.cs

The Google class acts as an interface between the jukebox application and the Google search engine. The principle function of this class is the "did you mean" feature from Google. Google does not have a webservicex api call to get this information. This class works simply by sending a normal search request to Google, the class will then scan the document that gets sent back to us, if it finds the did you mean phrase on the page, it calculates the offset and returns the phrase Google thought you meant. If it does not find this phrase on the page then it simply returns your original request string.

Guimain.cs

This is the front page of the jukebox. Most of the jukebox functionality is initiated from this class. The class is split into 5 main sections. At the top there are all the event handlers. These fire when a user initiates some action. Most event handlers call another function to carry out the task. This is done to ensure exactly the same semantics where there are more than one ways of initiating some functionality. In a very few number of cases where the functionality is very trivial and unique to that event handler, the functionality is executed within the event handler.

The next part of this file is the gui consistency methods. If any part of the program wants to activate or deactivate certain features or playback functions then calls should be made through these methods rather than directly to the widgets themselves. The purpose of this is to stop the user interface from getting into an inconsistent state.

The third part is two small functions for requesting dialog boxes for the user to add tracks.

The fourth part of this file is the audio playback functionality. There are two types of methods in this part of the file, methods which the gui should call and those which it shouldn't call. The methods which should and shouldn't be called are well documented within the guimain class.

Typically the methods that the gui should call will keep the gui in a consistent state. These methods will observe the current gui state and make changes to keep the gui state relevant. Those methods which the gui should not call generally have little effect on the gui, if they are not used properly then they could put the gui into an inconsistent state, potentially leading to a crash or lockup. A task for a later date is to extract this functionality into a new class. This functionality was originally in a different class but had to be incorporated into the main gui class for various reasons.

The fifth part of this file deals mainly with the user interface. The methods in this part are generally responsible for populating the widgets in the jukebox application. Additionally this is where the next and previous tracks are calculated. This is a poor design choice as it is dependent on running up and down the treeview widget to find the next tracks and previous tracks. There are also flaws which

means if a trunk is selected rather than a leaf, moving backwards or forwards will move you to the next trunk rather than the next leaf. A much better way of selecting next and previous tracks was partially implemented but not completed due to time constraints.

Id3v1.cs

This class will take an mp3 file and determine if there is an id3v1 tag resident on the file, if there is then the class will extract the relevant information and populate the variables in this instance. This information can be accessed via getter methods. The static offsets at the top should never need changed as the layout of an id3v1 tag never changes and is identical in all mp3 files.

Id3v2.cs

This class will take an mp3 file and determine if there is an id3v2 tag attached to a file. If a tag is found then various parameters need to be calculated as the structure of an id3v2 tag varies from mp3 to mp3. Once constructed, the tag information can be obtained via various getters. Whilst the structure of an id3v2 tag changes, the header is identical for every id3v2 tag. You should never need to change the header constants for the parser.

Jukebox.cs

This is the main class which starts the jukebox.

Library.cs

The library class takes care of reading and writing files into the library. When the jukebox is running it keeps track of files in sortedList containing the filenames as the key, and a track object as the payload. Consequently when reading and writing from the jukebox, it's the sorted list data structure that is passed via the methods. Files can be loaded simply by using a string path or, by an array of string path's if you are adding many files at the same time. The jukebox is stored in persistent storage as an xml file. There is a schema for that xml file encapsulates in the librarscheem.cs class

Librarscheema.cs.

This class was created to store the library schema within the program. A problem arises if a user copies the application but not the library schema. The presence of this class resolves this issue, because if the jukebox cannot find a schema where it expects one, it simple consults this class and writes a new one so that it can continue.

Playlist.cs

The purpose of the playlist class is to take the jukebox library and populate a treenode element that can be added to the treeview widget in the jukebox. The treeview widget allows users to browse and select tracks. A treenode can have children treenodes added to them, all treenodes except the leafs are dummy treenodes that do not have any information attached. The leaf nodes contain track information. The treenodes can be populated in any order, for example, album artist, or artist album. To determine which way the treenodes will be structured dsmply populate an arraylist

containing strings of the categories that the treeview can be ordered in. For example, album, or artist. Remember that the algorithm is case sensitive.

Stringtuple.cs

Stringtuple is a simple data structure that contains 2 strings. The purpose of this data structure is to allow a method to return 2 strings rather than just one.

Track.cs

The track.cs class acts as an interface between the jukebox and the parser. This exists to separate the parser from the jukebox. This class will coordinate the reading of metadata files for which you have implemented parsers for. If at any time in the future you wish to add support for parsing different types of files then this is the class that you should change. This class also has a dummy mode whereby it is desirable to add an object to a data structure that holds no useful information.

Trackmatcher.cs

The track matcher class attempts to determine if two tracks are actually the same track. Spelling and wording can vary from album to album and relying on exact matches substantially reduces the effectiveness of the music recommender. This problem is solved in two ways. Firstly the minimum edit distance algorithm is used to score the difference between two strings. A discretion percentage is given to suggest how much of the word is allowed to be spelt differently for it still to be considered the same word. Secondly there is a method that will allow two phrases to be considered the same even if one phrase is missing a couple of words. For example "the song title" and "the song title (radioedit)" should be considered as the same song.

Youtube.cs

The YouTube class contains methods for making a request to the YouTube web service. Any part of the application that wants to present the YouTube functionality to the user should instantiate a copy of this object using the argumented constructor. There is also a non argument constructor. This should be used if you don't want a gui. The class contains methods for searching for videos, this functionality is accessed via the listbytag method which returns an array list of pointers to videos.

Youtubegui.cs

This class deals mainly with gui issues relating to the youTube functionality. The processrequest method will take a search criteria string and populate the list box. The create page method is needed because of a bug in either the web browser widget or the youtube webpage, which means webpages, especially those containing flash, cannot be directly loaded from the webpage. To get round this we create a local html document and load this instead.

Youtubehack.cs

This class deals with a bug in the web browser widget. Normally when a mouse button is pressed on a flash application in the web browser, the mouse button gets stuck in the pressed state. This means that further clicks cannot be made on the flash application. This hack interrupts the windows message loop and resets the mouse button to its un-pressed state.

Youtubepreferences.cs

This class is a simple dialogue box that allows the users to change how many videos the jukebox should retrieve when the user makes a YouTube request.

Youtubevideo.cs

The youtubevideo class permits storage of certain YouTube video attributes. The class will also populate those attributes automatically if given an instance of the xml reader which is wound forwards to the first tag of the YouTube video in the xml document.

5. Known Bugs

The first known bug, and perhaps the most annoying one is that if the user selects a node on the treeview which is not a track, For example a dummy branch which might contain the artist name or album name only. If the user has selected such a node and then skips to the next track, or the track ends, the jukebox won't find the next track and will stop. The user will have to manually select another track. I attempted to rectify this but ran out of time. Current the jukebox calculates the next track by inspecting the treeview widget. By the time that I realised this was a bad idea it was too late. The new idea I have for fixing this is to only use the treeview only for getting information from the user or displaying information to the user. Instead of using the treeview to work out which track is next, I would use the library to work out which track is next making a request to update the treeview each time a track is played.

The second known bug is down to the handling of files that the jukebox expects to find but the user has deleted. Ideally the jukebox would just move onto the next track. It does not do this because playing a non existent or corrupt file does not fire an on audio ending event meaning that the jukebox never knows to start playing the next track.

The third bug is that the Amazon music recommender might stop's working altogether if the get recommendation button is pressed too much in a short period of time. I suspected that it might be the firewall on the computer that I was developing on however I have no way of quantifying this. If the jukebox gets into a state whereby it cannot process Amazon requests, simply close the application and start again.

I am aware that when the user presses the play button, the jukebox plays the first track in the sortedList library, rather than the first track on the treeview. I do not consider this a bug, This was the best hat could be done in the limited time available

There is one trivial bug that is out with my control. I cannot lock the graphical user interfaces so that the user cannot resize them. If the user resizes the jukebox then it can give an ugly appearance. Even with the lock enabled, the user is still free to resize the window.