Comp 2404 Fall 2017 Assignment 1

Due Date: Tue. Oct. 10 by 10:00pm in Culearn.

Assignment Submissions Instructions: Assignments must be submitted via culearn by the due date and time. Late assignments will not be accepted. See the Development Process Requirements below for what exactly to submit and how.

You may work on assignments in pairs if you want, in which case only one copy should be submitted with both your names on it files. If neither submits, because they thought the other one would, both get a mark of 0.

Revisions:

Rev 3 (2017 09 28): Some important customer feedback has been added below which comes from early prototypes revealed to them.

Rev 2 (2017 09 24) Added to requirement FR1.15 regarding comments. There is a clarification from the client regarding id fields in the data model. Also some minor typos.

Rev 1: fixed minor typos FR1.16

Categorization of Assignment Requirements

All assignments will be based on itemized requirements. There will be four categories of requirements: Process Requirements, Functional Requirements, System Requirements, and Domain Requirements.

Process requirements (aka Software Engineering Requirements, Development Process Requirements, Good Practice Requirements) are requirements imposted by the software development process that is being followed by the developers. Here they include the requirements on how to submit your assignments sucessfully and have them marked. They do not have anything to do with the specific application being developed, but rather pertain to the development process that will be used for the course. They include the requirements that reflect how your assignment will be tested and marked. In assignments you will not get marks for meeting process requirements, instead you lose marks if they are not met -in some cases ALL of your marks.

Functional, **System**, and **Domain** requirements all pertain to the actual application being developed. Collectively these are referred to as the **Design Requirements**. In assignments you will get marks for those design requirements that are met. So the number of these requirements determine what the assignment will be marked out of.

Functional Requirements (aka User Requirements) pertain to what the user wants to do with the app and how it is to be operated. If a functional requirement is not met one would expect a user or client to notice that. Functional requirements should make sense to the intended user or client. They should not contain references to "objects" in the object-oriented programming sense.

System Requirements (aka Constraints) are requirements imposed on the developer to make the development profitable (or evolvable, or resuable, ...). If a system requirement is not met the user, or client, of the app would likely not be aware of that. For example, the user of your app is not aware of whether you used objects to implement it or not. System requirements can make reference to "objects" in the object-oriented programming sense because the developers understand that.

Domain Requirements (aka Legal Requirements) are requirements imposed by the domain, or demographic, in which your app is intended to operate. For example, if you are making a PG13 movie about object-oriented development then the requirements that must be met for a movie to be considered PG13 are domain requirements. (Treating people as objects might be considered in appropriate for a young audience; treating objects as people will likey garner an R-rating.)

It will be an ongoing challenge in the course to fit requirements into one of the four categories. In real life there are often many more catagories.

Assignment Evaluation and Marking:

This assignment is based on 20 specific design requirements SR1.1... FR1.16 and we will be awarding 2 marks for those you are able to satisfy for a total of 40 marks. (Domain requirements are excluded for this assignment as they don't apply yet.) You will get 2 marks for each design requirement that is satisfied, well implemented and demonstrated through your testing output. You will get 1 mark for any design requirement only partly met, or met but not well implemented or not demonstrated in your testing. You get 0 for any requirement not satisfied. Conversely you lose marks for any Process Requirement not met as indicated below.

Development Process Requirements

The following requirements will pertain to all your assignments regardless of what your application is supposed to do (i.e. regardless of the design requirements). These requirements are intended to ensure that your code is readable and maintainable by other programmers (or markable by TA's in our case), robust (it does not crash from bad pointer references or memory allocation problems), and following good object-oriented programming practice. You will lose 5 marks from your total assignment mark for each of the following requirements that is not satisfied. However if you do not satisfy requirement PR0.1, PR0.2 or PR0.3 you will get zero for the assignment mark.

PR0.1)[Assignment mark = 0 if not met] IMPORTANT Uniqueness Requirement. The solution and code you submit MUST be unique. That is, it cannot be a copy of, or be too similar to, someone else's assignment, or code found elsewhere. A mark of 0 will be assigned to any assignment that is judged by the instructors or the TA's not to be unique.

(You are however free to use any code posted on our course web site as part of the course notes or example code provided in the notes, assignments or tutorials.)

PR0.2) [Assignment mark = 0 if not met] CODE SUBMISSION REQUIRMENTS: For the purposes of assignments your code and supporting documents must be submitted to culearn and comply with the following.

1) You should submit C++ source code file and any supporting documentation files required and, when appropriate the associated make file to assist the compilation. Your name and student number should be at the top of each file. There should always be a ReadMe.txt file with your assignment. If you are working with a partner, both names and student numbers must appear on the files and only submit one copy of the assignment to culearn.

2) COMPRESSION If your assignment files are compressed we will accept only .tar, .tar.gz, or .zip format compatible with one of the following three linux commands for extraction:

tar -xvf filename.tar
tar -xvf filename.tar.gz
unzip filename.zip

3)ReadMe.txt: Your assignment must be accompanied by a ReadMe.txt file in which the following should appear:

-Which extraction command should be used to uncompress your assignment files.
-Which g++ compiler command should be executed to compile your code.
-Instructions on what script, or scripts to run to demonstrate your testing.

4) In general you should provide a testing script, or main program that runs appropriate test cases to demonstrate that the design requirements are being met. (Not all requirements can be demonstrated like that, but functional requirements usually can be -especially if we are building command line applications.)

PR0.3) [Assignment mark = 0 if not met] CODE COMPLILATION and TESTING:

The TA must be able to compile your code using the g++ compiler installed in our 2404-240-F17.ova virtual linux image. (A

current gcc installation should be compatible with that, but you should check.). The TA will run the g++ command specified in your ReadMe.txt file and when compiling is complete run your exectuable to begin verifying the assignent requirements. If your code cannot be compiled as described the assignment mark will be zero. (The TA's will not debug your code in an effort to get it to compile.)

PR0.4) CODE ORGANIZATION: You code files should contain only one .cpp file containing the substring "main" in the title and that file should have the `int main()` entry point function for your code. Your object-oriented classes should each be represented in two files `classname.h` and `classname.cpp`. The .h header files should contain only type declarations and not code definitions (method bodies).

PR0.5) VARIABLE AND FUNCTION NAMING: All of your variables, methods and classes should have meaningful names that reflect their purpose. Do not follow the convention in math courses where they say things like: "let x be the number of customers and let y be the number of products...". Instead call your variables `numberOfCustomers` or `numberOfProducts`. Your program should not have any variables called "x" unless there is a good reason for them to be called "x". (One exception: it's OK to call simple for-loop counters i,j and k etc. when the context is clear and VERY localized.)

PR0.6) MEANINGFUL CONSTANT NAMES: Constant values should have meaningful names and not just be represented as numbers or strings in expressions. For example don't say:
```
cin.getline(input, 80);
```
rather say:
```
cin.getline(input, MAX_INPUT_LENGTH);
```

PR0.7) PRIVATE VARIABLES: All variables in your classes should be `private`, unless a specific design requirements asks for them to be `public` (which is unlikely). It is good programming practice to design objects that provide services to others through their `public` methods. How they store their variables is their own private business. C or C++ unbounded arrays should not be used as public data structures. They should be encapsulated in a class which takes care of their length or number of elements. Also methods used soley by a class for its own housekeeping, and which are not considered part of the public inteface, should be in the `private` section of a class.

PR0.8) ROBUSTNESS REQUIREMENTS: Your program should not crash when the TA is marking it because of a bad pointer dereferences, out of bounds error memory access, or memory leaks or double deletions. We get especially annoyed by out of bounds "off by one" errors! Your code should not have any memory leaks or double deletions. Moreover the heap memory should be empty when your procedure `main()` returns. That is, all heap objects should be properly deleted before the program ends.

PR0.9) COMMENTING REQUIREMENTS: Comments in your code must coincide with what the code actually does. It is a common bug in industry for people to modify code and forget to modify the comments and so you end up with comments that say one thing and code that actually does another. Don't use comments to clarify poor variable or method names -instead choose good variable names and method names that makes the code more "self-commenting".

PR0.10) OUTPUT LABELING REQUIREMENTS: Your testing output that you hand in, or that results from the scripts you submit, must have sufficient comments or remarks so that the output can understood. Don't have your program just write out a bunch of integers with no indication of what they mean or what the test they supposedly demonstrate. The TA's should be able to read and understand your output WITHOUT having to look at your code to see what the output means. It is a good idea to refer specifically to the numbered design requirements in your output to help indicate which requirement you are demonstrating.

VERY IMPORTANT:

Requirements tend to make sense to the person who wrote them because that person knows what they were trying to say. You don't know what they were trying to say, only what they did actually say. So you need to ask lots of questions when things are not clear. A picture is worth a thousand words and a requirement fix is worth a thousand programing fixes.

Any sample code fragments provided with assignments might have bugs (although none are put there intentionally). It is part of your job to identify errors in the code (and in the requirements) and seek clarification.

Assignment #1 Design Requirements

Background

This development project pertains to the following scenario.

A household, or group, of people, want to combine their individual CD music collections by rippring them all to mp3's and then accessing the collection with a command line app. The app should house the music collection and provide users (members) with access. A user should be able to create playlists and place selections (songs) from the music collection in their various playlists. A user should be able to search for songs and add and delete songs from their playlists or to and from the music collection as a whole. (Ultimately a user should be able to play song selections but our initial development scope will be building an app to manage the information about the songs and provide access to that).

The client has tried some commercial mp3 players (e.g. itunes, amazing slow downer, anytunes etc.) and found them all to be unsatisfactory. Here are some of the "deal breakers" with other apps that should be addressed in this development.

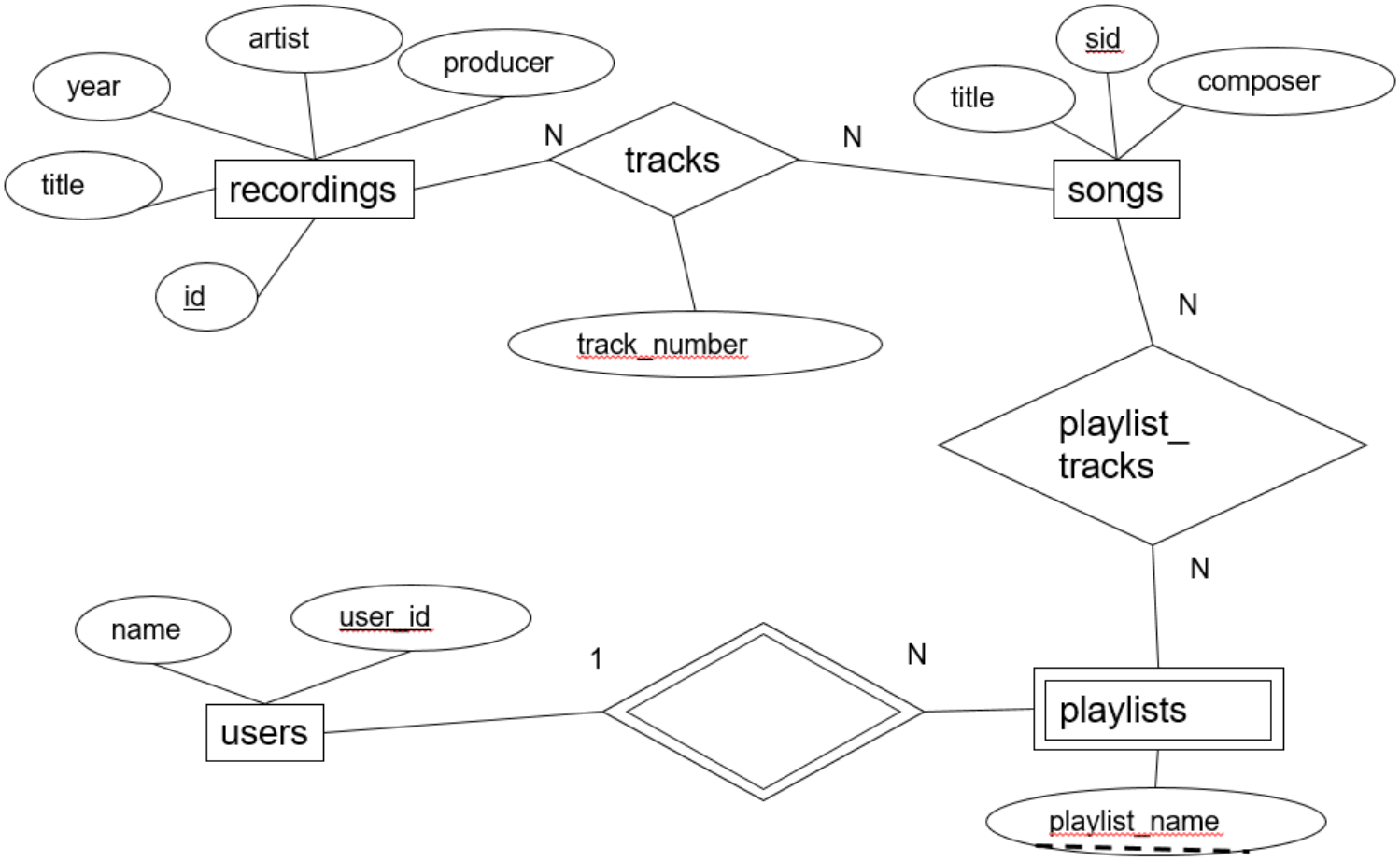
1) A user should be able to add and remove songs by way of the playlists and not just a by way of a global songs collection. When you delete a song from a playlist it should be, optionally, possible to delete it from the entire collection as well.

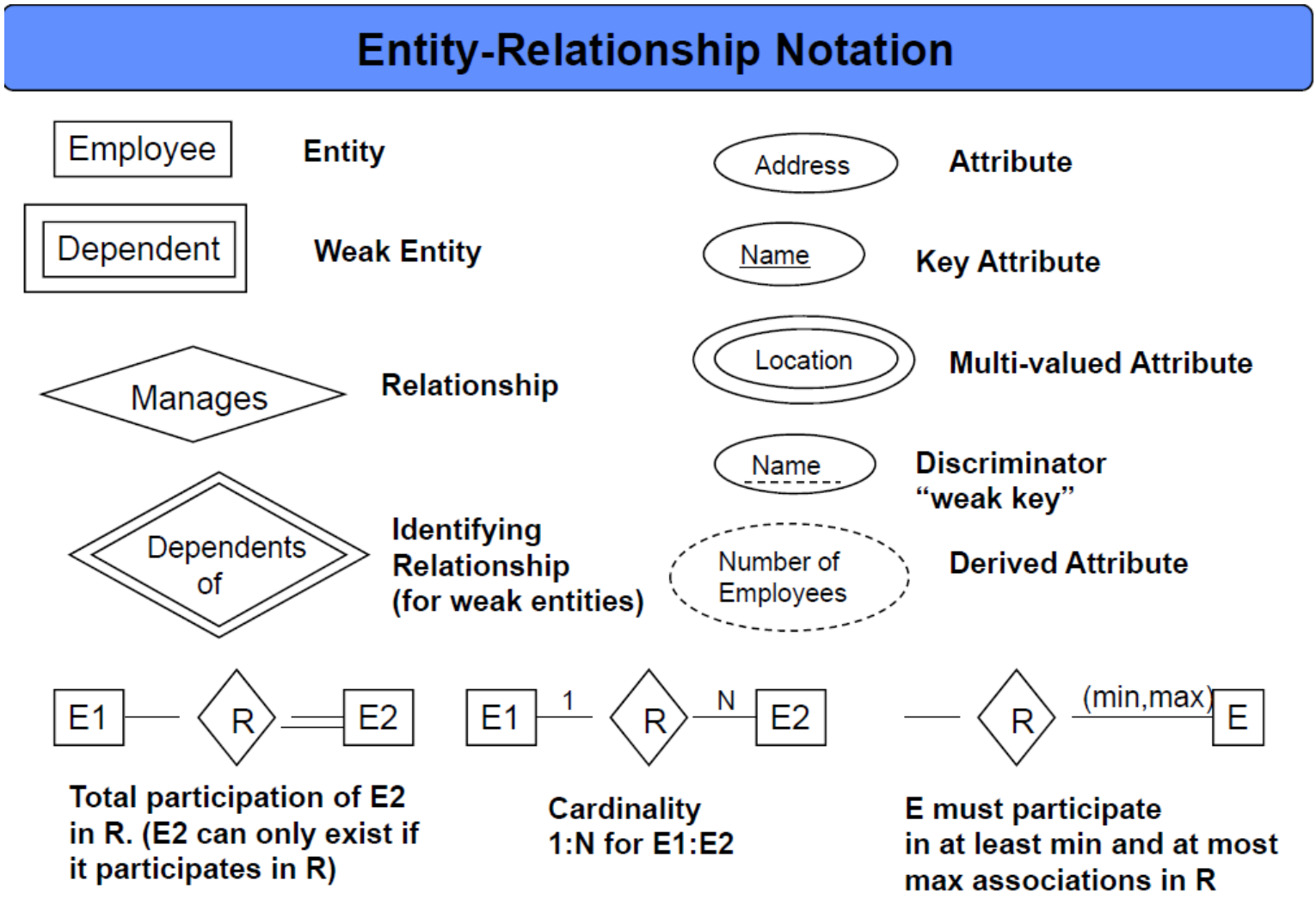
2) Searching for songs should not be encumbered by case or punctuation in song titles or CD titles. One should be able to search based on partial information about titles, names etc.

3) A user should be able to script commands and run them from a file and not just execute them one by one interactively on the command line interface.

4) A user should be able to log the commands and repsonses and save them as a record (or audit) of a session with the app.

The client has provided the following data model (expressed as a very common E-R diagram) and provided a sample sqlite database with some sample data that reflects what is presented in the E-R model). This data set consists of the songs from the studio albums of the Beatles and some users with playlists.


artist
producer
year
title
recordings
id
N
tracks
N
track_number
sid
composer
title
songs
N
playlist_
tracks
N
name
user_id
1
N
users
playlists
playlist_name



The sqlite database with the sample data is in the file `beatles.db` and can be accessed with the sqlite3.exe application by executing `sqlite3 beatles.db` in a command terminal.

Or (in Windows) by double-clicking on the `sqlite3.exe` file and then executing `.open beatles.db` on the commnad line.

The sample relational database has the following table schema:

recordings				
id	title	artist	year	producer

songs		
sid	title	composer

tracks		
albumID	songID	track_number

users	
user_id	name

playlists	
user_id	playlist_name

playlist_tracks		
user_id	playlist_name	song_id

In this assignment and others we will use the sqlite3.exe to investigate the client's data model and also it will serve as an example of a command line interface that works quite effectively. This database application will be demonstrated in lectures and will likely be the subject of one of the tutorials.

[Rev 2 Clarification regarding id fields in data model] The client has clarified that the id fields used to identify recordings, songs and link songs to recordings (i.e. tracks) have no significance to their format or values. They could be numbers generated automatically by a counter for example. The user_id field however is likely to be chosen by the user and could have meaning to them. Like the userid of an email account.

Scope

This development phase (assignment) has the following scope.

For this assignment you will design and implement the command language and "front end" command line interface that the user will use to operate the application. One of the requirements is that commands can be read from a script file and executed one at a time. Thus along with your code you need to submit two scripts that use the language that you design. One script should be the commands that would enter all the data from the beatles.db data set into your app. The other a script that illustrates that the functional requirements for the language have been met. Note we are only designing the front end here so we will not actually represent the data inside the app yet.

The following known requirements are being left out of this phase (assignment): searching for songs and recordings by title, authentication of members, actual representation of the data within the running app.

The main C++ programming capabilities you will likely need for the assignment are:

Writing prompt strings to the terminal.

Reading user command strings from the terminal.

Parsing a command string into its revelant command parts.

Opening and reading a text file one line at a time.

Creating and Writing to a text file one line at a time.

Storing strings in a container (to serve as the session log.) which can be written out to a file.

For this assignment you need to design the command language and build a working command line interface based on it. For this assignment you need to parse and "execute" the commands. Executing a commands at this stage will just produce "dummy" output to acknowledge the the command was properly received and parsed. The script reading and logging however must be made functional in this phase. Note it is not enough just to echo back what a user typed. Your output should demostrate that you properly interpreted the parts of the command (command name, arguments etc.)

You must provide a .txt scripts written in your designed command language that will, in theory, populate your app with all the data that is provided in the beatles.db database and another to illustrate that the commands comply with the functional requirements.

Be aware that your command language needs to be scriptable (one command per line like linux bash scripts) but your interface need not be. For example, your command line interface could expect the user to type the entire command, or it could accept just a keyword and then prompt the user for addtional attributes, or it could print a menu of choices and ask the user to make a selection. You need to design what you think will work best for the intended application and new requiremets you anticipate.

Also your interactive interface might have some commands the are "shell only" commands and not part of the commands that you might put in a script. The scriptable part of the language should contain all the commands that are necessary to get work done with the application, but there might be additional commands that only pertain to using the interactive command line. For example a "help" command, which shows available commands, might make sense to someone interacting with the interface but not when executing commands from a script file. Shell only commands should be distinguished from scriptable commands by starting with a "." dot. For example `.help` rather than `help` if help is only a shell command.

REV 3: IMPORTANT? INITIAL CUSTOMER FEEDBACK:

(Based on work shown to me during office hours, the customer has been consulted and shown several prototype outputs from the "help" command. One such is this prototype_help.txt. They have expressed the following reaction to the early prototypes:

NOTE THESE DO NOT CONSITUTE NEW REQUIREMENTS FOR ASSIGNMENT #1. They do however anticipate possible future (assignment #2) requirements. But remember: predictions are hard -especially the ones about the future.

This revision highlights some important questions regarding the design of a software engineering/software development process:

1) To what exent can the customer change their mind about requirements?

2) If customer feedback is going to be used how should it be accounted for and incorporated and when?

3) Is there a point at which the customer must "sign off" on the requirements?

4) Is this revision a sign of poor requirement gathering or were the prototypes a necesary step in obtaining better requirements? ("A man who picks a cat up by its tail learns something that can be learned in no other way" -Mark Twain.)

5)Should the customer have been denied access to the early development work or alternatively, should the customer feedback be kept from the development team during a development phase?

Here are the customer reactions to some prototype help command outputs:

1) There is a strong preference for multi-word data arguments to appear in quotes: e.g.
add song "The Way We Were" 1234
as opposed to:
add song: The Way We Were id: 1234

2) There is a preference for "-x" "bash-style" attributes to keep commands short, for example:
add -s "The Way We Were" 1234
as opposed to
add song: "The Way We Were" 1234

3) There is a preference for compound data over attributes identifying individual data items, for example:
add -s "The Way We Were" 1234
as opposed to:
add -s title: "The Way We Were" id: 1234

4)There is a strong preference for multiple word commands over compound words, for example
.log start ouput
over
.log start_ouput

5)There is a preference for multiple word commands to use words rather than "-x" style flags for example:
.log start output
over
.log start -o
or
.log -o start

6)There is a strong preference of commands that involve "-x" flags to allow them in any order, for example
add -s "The Way We Were" -u ldnel -p "Driving Songs"
to be equivalent to:
add -p "Driving Songs" -u ldnel -s "The Way We Were"

7) The customer appears very sensitive to Jacob's Law. Jacob's Law is a law of web-page (user-interface) design which states: "People will spend most of their time on other sites". The implication is that how someone tries to use your interface will depend on their experiences with other interfaces. On a smaller scale: how they try to use your "delete" command will depend very much on how they must use your "add" command.

Again, this feedback does not constitute new assignment requirements -yet.

Domain Requirements

(The following two requirements have been categorized as Domain requirements -though they look suspiciously like user requirements. Why might that be?)

DR1.1) Any song, CD, or Album titles or any band names should always be displayed for the user in title case. That is, the first letter of each word should be capitalized and any title or name that starts with the word "The" should be displayed with the "The" at the end. (e.g. The Beatles should be displayed as Beatles, The) [This will apply more to future assignments but is added now to show an example of a domain requirement.]

DR1.2) Any searches for song titles or recording titles should ignore both case and punctuation. For example "take the A train" should match "Take The 'A' Train".

DR1.3) Any searches for song titles or recording titles should accept placeholder characters repesented by a "*". From example "* Ipanema" would match "The Girl From Ipanema".

System Requirements (Constraints)

SR1.1) The application should be written in C++ and have a command line interface.

SR1.2) The application code should be fully object-oriented. That is, except for the `int main(){}` entry point function and maybe some initialization helper functions, all funtionality should be through **methods** of some `class` (either `static` or instance methods).

SR1.3) The application should decouple those objects used to implement the interface from those at represent the control logic. Also objects that represent domain entities (things from the client data model) should be decoupled from the interface and control objects.

SR1.4) Entity objects (that pertain to the data model) should be represented throught a single instance and not copies. For example, if a song is an object and it appears on several playlists, the playlists should refer to a single instance and not have separate copies of the song object. Playlists, for example, could "point" to song objects or could contain state information, like id or title, that could be used to identify the actual song object. This is very much in line with what is considered the "object-oriented" way of doing things.

Functional Requirements (User Requirements)

Command Language Requirements

Note in the descriptions below commands that appear in "" quotation are meant only to convey what the command might do and are not meant to suggest what the command name or syntax should be. Also an important design decision is whether a particular command is a "script-able" command or only pertains to using the interactive command line.

The TA's will be testing your interface butwill do so only after running the two scripts you are required to provide: one that will enter all the data from the beatles.db database and another to demonstrate that the functional requirements are being met (e.g. one that illustrates each command you are responsible for.) Without these scripts the requirements below will be deemed not to have been met.

FR1.1) The command language syntax should always start with command name followed by whatever arguments are required for the command to execute. (Not unlike the terminal commands in linux or DOS shells). The commands should be scriptable (one complete command per line of text) but the interactive interface could use multiple lines and prompts. Commands that are "command shell only" should start with a "." dot. For example, if `help` is a shell only command the command name should be `.help`. The arguments of a command should be separated by blanks.

FR1.2) When the application launches it should present the user with a command line terminal and prompt ready to receive commands. When a command executes the results should be displayed for the user followed by another prompt and be ready to receive the next command. (Again not unlike the bash terminal in linux or DOS).

FR1.3) The application should have a "help" command to show, or remind, the user what the available commands are.

FR1.4) The application should have "display" commands to display the various collections: recordings, songs, users, users's playlists etc.

FR1.5) There should be commands to display what playlists a particular user owns.

FR1.6) There should be commands to display what songs appear on a partiular playlist of a user.

FR1.7) There should be "add" commands to add new recordings, songs, user, or playlists.

FR1.8) The commands should allow establishing all the relationships that exist in the data model. For example: associating songs with recordings, or playlists with users, or songs with user playlists.

FR1.9) The application should have "delete" commands to delete recordings, songs, users, or playlists. Also the application

should preserve referential integrity if items are deleted. For example is a song is deleted from the application then it should also be deleted from any playlists it appears on.

FR1.10) The commands should allow the deletion of songs from a playlist to only affect that playlist or optionally permanently deleting the song from application altoghether (including from other playlists on which might appear).

FR1.11) The command line interface should include a shell only "read" command that will allow a user to read and execute commands from a script file. The script file would be expected to have one scriptable command per line. Once a script file has been read the interface so go back to the interactive REPL mode.

FR1.12) Since the application provides for reading a script of commands, it should be possible to create the entire data set with a script file of commands. That is, load the application on startup from a script file written in the same command language.

FR1.13) There should be a "logging" command that allows users to start and stop the logging of executed commands. Logging would record the commands the user is executing and allow them to save an audit of those to a file.

FR1.14) The "logging" command should allow users to log only the commands executed but not their responses, or both the commands and responses. [Notice with the logging functionallity it would be easy to make testing output.]

FR1.15) There should be a "comment" command that would allow a user to add comments or remarks to their console session or log output. This could be useful for inserting remarks within a logged session explaining what the commands are doing or testing. Also [Rev 2] it should be possible to have comments in the script files that are executed with the read command [see FR1.11].

FR1.16) Two scripts must be included with your submission, written in the designed command language. One that would initialize the application with the complete data of the beatles.db sample database and the other to demonstrate that all required commands have been implemented.
