**FR3.2.1) Identify the aspect of the code you DON'T like and describe what it is. Provide some sample lines from the provided code to make it clear what you have identified.**

1. In *mytunes.cpp* the code provided in *executeCMDSHOW(*) method was checking the same condition twice in the if statement as following:

|  |
| --- |
| if(cmd.isTokenAt(COLLECTION, "songs") && !cmd.hasToken("-s")){  songs.showOn(view);  }  else if(cmd.isTokenAt(COLLECTION, "songs") && cmd.hasToken("-s")){  songs.showOn(view, cmd.getToken("-s"));  } |

This piece of code was improved by using a “common” condition in if statement – this way it checks for the condition only once instead of twice. Also, it makes sense to use else statement since the conditions are opposite and the program only executes cmd.hasToken("-s")) once, as shown below.

|  |
| --- |
| if(cmd.isTokenAt(COLLECTION, "songs")){  if (!cmd.hasToken("-s")){  songs.showOn(view);  }  else  songs.showOn(view, cmd.getToken("-s"));  }  } |

1. In the given code, Recordings, Songs, and Tracks classes supported member lookup by numeric id using a findByID() method whereas Users class uses string instead in findByUserID() method. It could be improved by having one uniform method that all the collections can use. The refactored code is explained further in **FR3.2.3**.
2. Common methods were used multiple times in different collections. So that needed to be refactored. The refactoring strategy is explained under **FR3.2.4** in this document.
3. I also didn’t like how the provided code doesn’t display to the user more information regarding the execution of the input command for e.g when the user wants to add a song, it doesn’t explicitly print out something like, “Song with title “Blab la” has been added” and then now the collection size is “some integer value”. It’s not a big of a deal but I personally think I would provide this information like I did in my submission for assignment 2.

**FR3.2.2) Provide an explanation of what you find objectionable about the code you've identified and explain why you think it should be refactored.**

1. In the given code, toString() function wasn’t being used.
2. It looks like void showOn(UI & view, const string & memberID)wasn’t being called for any of the inputs that are provided in the helpfile.txt
3. Collection-based classes (Recordings, Songs, etc) have almost all the methods in common which leaves us with a really good reason to refactor the code.

|  |
| --- |
| template <typename T>  class MyTunesCollection{  public:  MyTunesCollection(){}    ~MyTunesCollection(){  for(int i=0; i<collection.size(); i++){  delete collection[i];  }  }  void add(T &aT){  collection.push\_back(&aT);  }  //similar common methods follows |

**FR3.2.4)** **Explain your refactoring strategy and how you intend to refactor the code. Provide some sample code or pseudo-code to illustrate your strategy.**

Part 1 Refactoring

The common variable and methods that were implemented in corresponding collection-based classes (Recordings, Songs, Tracks, and Users) were put in a single template file named *mytunes\_collection.h*. collection vector was the only common variable used in the collection-based classes, and hence was made a private member of the generic class, as shown below:

|  |
| --- |
| template <typename T>  class MyTunesCollection{  public:  MyTunesCollection(){}    ~MyTunesCollection(){  for(int i=0; i<collection.size(); i++){  delete collection[i];  }  }  void add(T &aT){  collection.push\_back(&aT);  }  // the rest of the “common” methods aren’t shown here  private:  vector<T\*> collection; |

To address the issue of toString()method, an << overloading operator was created that takes in the reference of the collection (Songs, Recordings, etc), calls printOn() method in which toString() method is invoked on the corresponding instance (Song, Recording, etc). In addition, getClassName() method was created in UI() class, which returns the string that contains the name of the class, which will be outputted for show commands. Below are the code snippets (from different classes) that show the modification made to the code.

|  |
| --- |
| String getClassName()){  Return collection\_name;  }  void showOn(UI & view) {  cout <<"Showing ";  view.printOutput(view.getClassName());  cout <<\*this<<endl;  }  void printOn(ostream &o) {  for (int i=0; i<collection.size(); i++){  string uh = collection[i]->toString();  o << collection[i]->toString()<<endl;  }  }  template <typename T>  ostream &operator<<(ostream &o, MyTunesCollection<T> &t){  t.printOn(o);  return o;  } |
|  |

Part 2 Refactoring

In part 2, a super class (Collection) was created that included the common methods used in classes i.e. Song, Recording, Track, and User. In other words, the mentioned derived classes inherit from the Objectollection super class, guaranteeing that classes maintain a certain common interface that is, they implement the same methods. The common methods that were included in the super class are shown in the code snippet below The variables were made protected so they could be used in derived classes and hence the common methods ie getID(), getUserID() and the included variables were commented out from the corresponding derived classes. toString() method was made pure virtual since its implementation is different in the derived classes. Alternatively, it forces the derived classes to implement this method too thus making ObjectCollection class abstract.

|  |
| --- |
| class ObjectCollection{  public:  int getID(){  return id;  }    string getUserID(){  return userid;  }    //pure virtual function  virtual string toString() const = 0;  protected:  int id;  string userid;  }; |

**FR3.2.5)**

Below are the UML diagrams before and after refactoring the code respectively.

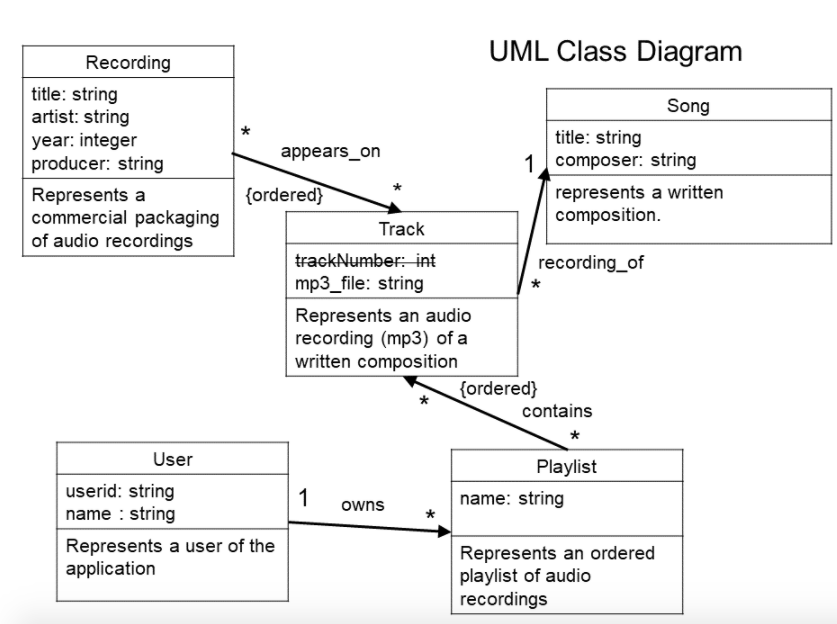
­­

Figure : UML diagram before refactoring the code

We can see the ObjectCollection is the super class whereas Recording, Song, Track and User are the derived classes.

Figure : UML diagram after refactoring the code

