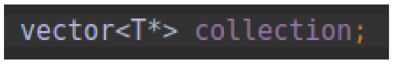
FR3.2.1) The aspect that my partner and I didn't like in the code, was the linear time complexity of working with vectors, O(n). Since MyTunes is an app that works with collections that could be very large, the actions of adding, finding, and removing could become quite expensive as the collections grow.



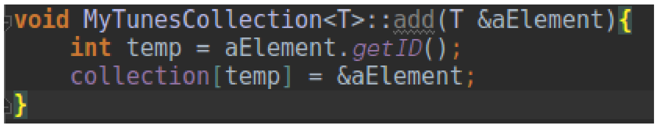
```
vector<T*> collection;
```

FR3.2.2) We found the use of vectors objectionable because there are other data structures that would work just as well as vectors but would operate significantly more efficiently. For example, using a map instead of a vector would improve the speed of functions that add or remove elements to/from the collection. These functions have O(n) when used with a vector, but improve to O(log n) when used with a map.

FR3.2.3) Our refactor does make a real improvement. For example, if the collection "songs" is a vector and has 1,000,000 elements in it, it has O(1,000,000) complexity worst case scenario (when adding or removing). However, with a map containing 1,000,000 elements, add/remove are reduced to having O(6) complexity. This is a massive improvement in efficiency and is definitely worth doing in our opinion.

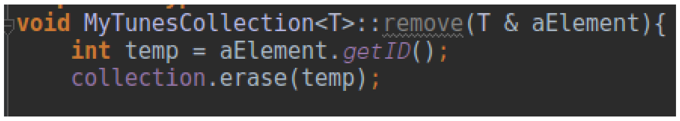
FR3.2.4) Our refactoring strategy is fairly straightforward. First, we replaced the type of data structure that MyTunesCollection.h used to store elements in (from vector to map). Then, we adjusted how the add, remove, findByID, and showOn functions worked in the MyTunesCollection.h template in order to work with maps. The functions add and remove actually became much simpler and easier to read. For findByID, we chose to overload the function to accept either an integer or a string as a parameter.

Add:



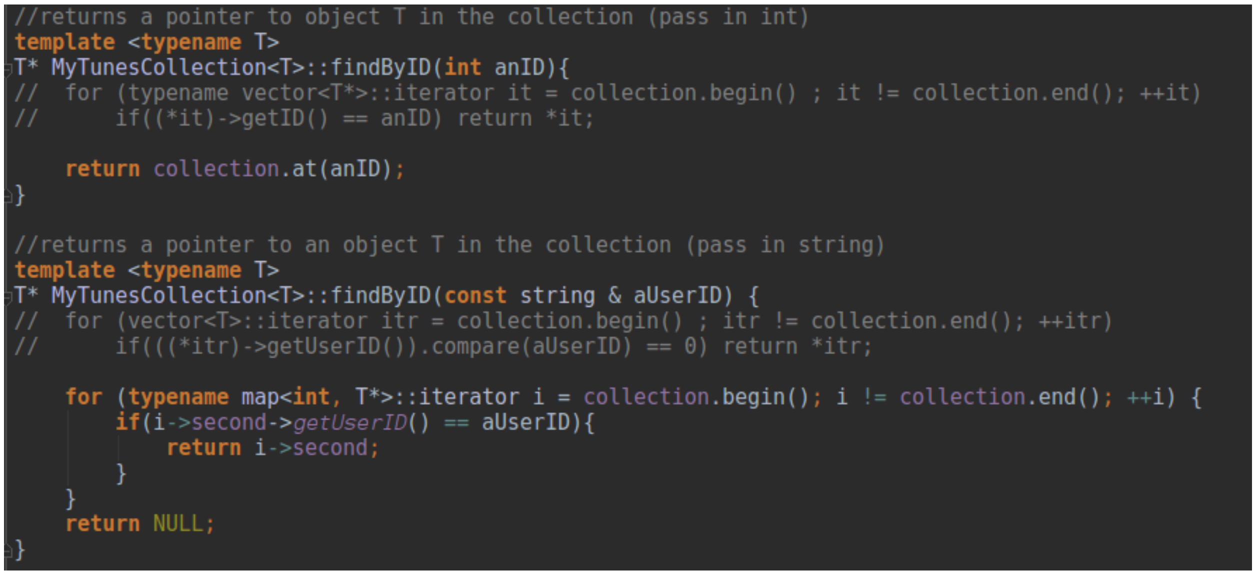
```
void MyTunesCollection<T>::add(T &aElement){
    int temp = aElement.getID();
    collection[temp] = &aElement;
}
```

Remove:



```
void MyTunesCollection<T>::remove(T & aElement){
    int temp = aElement.getID();
    collection.erase(temp);
```

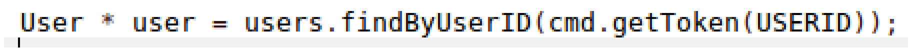
findByID:



```
//returns a pointer to object T in the collection (pass in int)
template <typename T>
T* MyTunesCollection<T>::findByID(int anID){
//  for (typename vector<T*>::iterator it = collection.begin() ; it != collection.end(); ++it)
//      if((*it)->getID() == anID) return *it;

    return collection.at(anID);
}

//returns a pointer to an object T in the collection (pass in string)
template <typename T>
T* MyTunesCollection<T>::findByID(const string & aUserID) {
//  for (vector<T>::iterator itr = collection.begin() ; itr != collection.end(); ++itr)
//      if(((*itr)->getUserID()).compare(aUserID) == 0) return *itr;

    for (typename map<int, T*>::iterator i = collection.begin(); i != collection.end(); ++i) {
        if(i->second->getUserID() == aUserID){
            return i->second;
        }
    }
    return NULL;
}
```

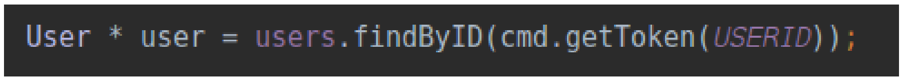
(we left the original code, to demonstrate the differences, and how much simpler our code was)

We also had to make a few adjustments to mytunes.cpp in order to implement the template properly. For example,



```
User * user = users.findByUserID(cmd.getToken(USERID));
```

became,

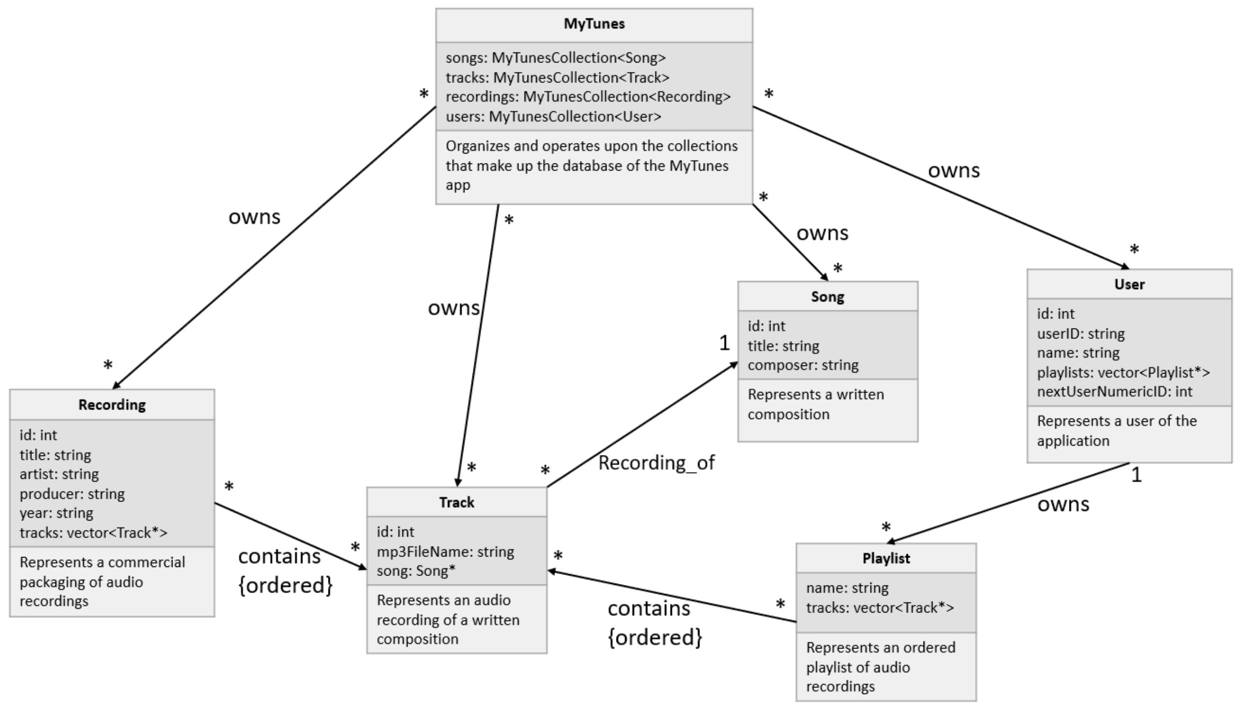


```
User * user = users.findByID(cmd.getToken(USERID));
```

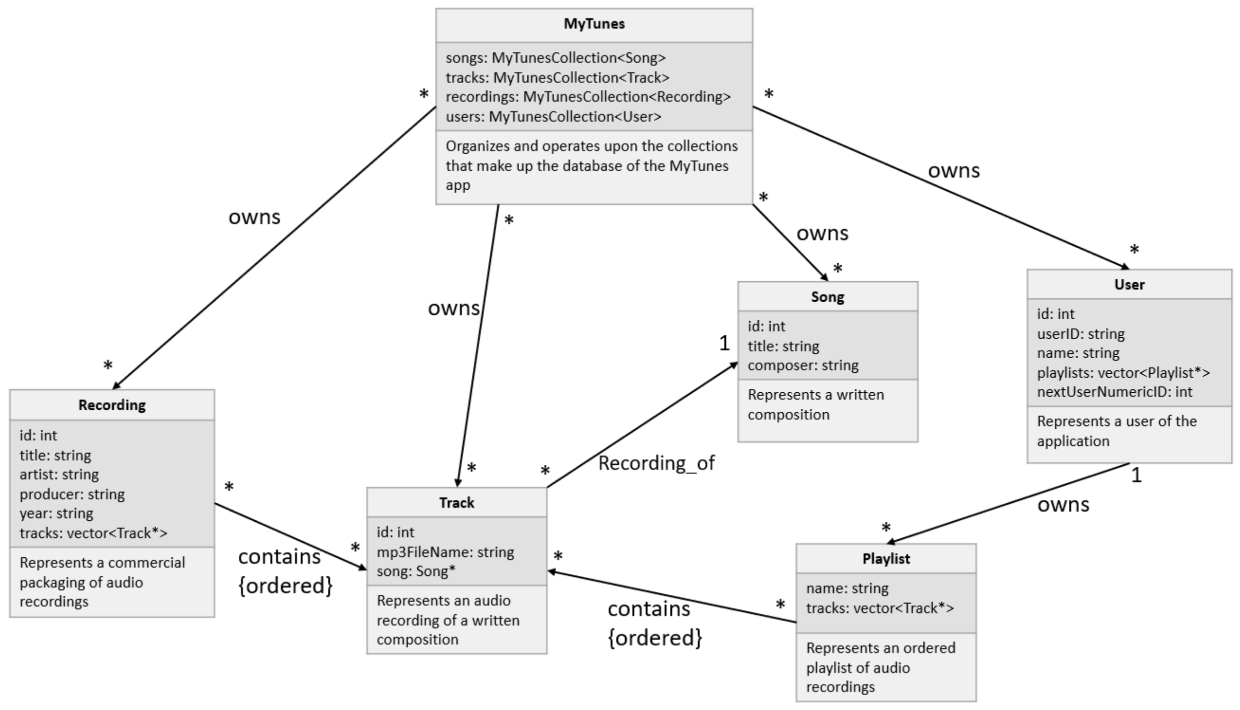
in order to use the overloaded findByID function.

FR3.2.5)

Before:


MyTunes
songs: MyTunesCollection<Song>
tracks: MyTunesCollection<Track>
recordings: MyTunesCollection<Recording>
users: MyTunesCollection<User>
Organizes and operates upon the collections that make up the database of the MyTunes app
owns
owns
owns
owns
Song
id: int
title: string
composer: string
Represents a written composition
User
id: int
userID: string
name: string
playlists: vector<Playlist*>
nextUserNumericID: int
Represents a user of the application
Recording
id: int
title: string
artist: string
producer: string
year: string
tracks: vector<Track*>
Represents a commercial packaging of audio recordings
Track
id: int
mp3FileName: string
song: Song*
Represents an audio recording of a written composition
Recording_of
contains {ordered}
contains {ordered}
owns
Playlist
name: string
tracks: vector<Track*>
Represents an ordered playlist of audio recordings
*
1

After:


MyTunes
songs: MyTunesCollection<Song>
tracks: MyTunesCollection<Track>
recordings: MyTunesCollection<Recording>
users: MyTunesCollection<User>
Organizes and operates upon the collections that make up the database of the MyTunes app
owns
owns
owns
owns
Song
id: int
title: string
composer: string
Represents a written composition
User
id: int
userID: string
name: string
playlists: vector<Playlist*>
nextUserNumericID: int
Represents a user of the application
Recording
id: int
title: string
artist: string
producer: string
year: string
tracks: vector<Track*>
Represents a commercial packaging of audio recordings
Track
id: int
mp3FileName: string
song: Song*
Represents an audio recording of a written composition
Recording_of
contains {ordered}
contains {ordered}
owns
Playlist
name: string
tracks: vector<Track*>
Represents an ordered playlist of audio recordings
*
1

By changing vectors to maps, no UML changes were necessary.

FR3.2.6) and **FR3.2.7)** The refactored code is part of the assignment and the app behaves the same as originally

FR3.2.8) As stated in the ReadMe.txt file, the script file `Part2TestScript.txt` will execute all of the commands the app has to ensure that everything is working correctly after the refactor.