Audio Engine Design Document - GDSoundEngine

Seinfeld

by Kerwin Ghigliotty Rivera

Contents

High Level Overview

The goal of this project is to create an Audio Engine derived from Win32's XAudio2 called GDSoundEngine.

The user would be able to Load a particular playlist and perform any specific command on it (Play, Pause, Stop, Raise/Lower Volume, etc). This should be accomplished with the use of an audio thread, as well as a File thread and a Helper thread.

As of this iteration, the following actions are initiated on the Game (main) thread and passed onto the Audio thread as part of commands:

- Load Files – Calls a derived version of Win32 File functions called FileSlow, which simulates slow loading. The command to load files is then delegated to the File thread for loading, we can use this in two different ways:
    - Blocking – The game thread is stopped until the audio finishes loading, an internal callback is then passed from Audio to Helper thread and when the audio finishes this gets triggered and the blocking exits.
    - Asynchronous – The game thread continues and if the user specifies a LoadCallback this will be passed onto the Game thread to let the user know when the load finishes.
- LoadPlaylist – Creates a Playlist Reference node in the PlaylistRefManager, this makes it available to be instanced with the Add command.
- AttachVoice/Command – Attaches a Voice or Command to the specified Reference playlist
- StartPlaylist – Sends the commands of the playlist to the TimerEventManager to begin execution of all the attached commands.
- Play – Starts the playback of the main voice.
- Pause – Pauses the playback of the main voice.
- Stop – Stops playback of the main voice.
- Raise/Lower Volume – Raise or lower the volume by a set default value.
- Set Volume – Sets the volume to a specified amount.
- Raise/Lower Pitch – Raises or lowers the pitch by a set default value.
- Pan Left/Right – Pans the audio towards the left or right channel.
- Set Pan – Sets the pan to a specific value [-1,1].

How it works:

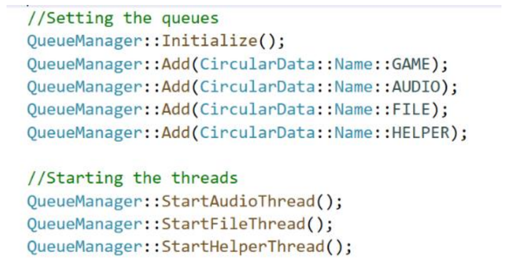
Instructions are sent to CircularQueues, these CircularQueues represent their respective threads, for this engine we need four specific threads that are initiated on detached mode:

- Game thread (or Main thread) – This thread by nature will control the GDSoundEngine which handles the Sound and Queue managers.
- Audio thread – this thread handles all the audio actions as well as handle the delegations to the File and Helper threads. It oversees the ASound, Voice, TimerEvent, Playlist and PlaylistRef managers.
- File thread – this thread performs the load operations and takes care of the File Manager.
- Helper thread – this thread performs the update to internal tables, File table as well as Priority table.

GDSoundEngine sends instructions to the queues associated with these threads and these threads are all managed by the QueueManager. This QueueManager handles the initiation of these threads as well as their shutdown. The instructions sent by the GDSoundEngine are then consumed by the threads and disposed of.

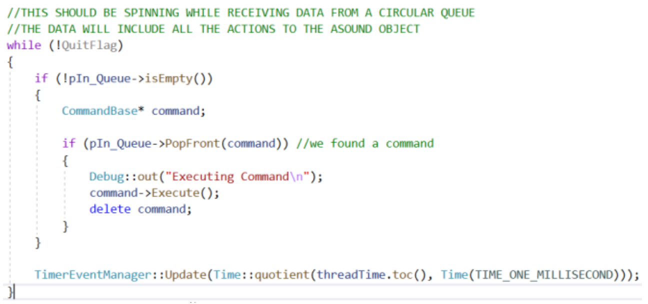
Threads Being launched

(This is part of the GDSoundEngine)

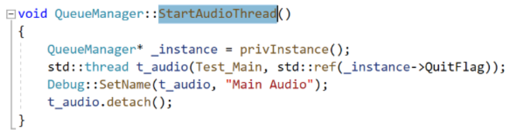


```
//Setting the queues
QueueManager::Initialize();
QueueManager::Add(CircularData::Name::GAME);
QueueManager::Add(CircularData::Name::AUDIO);
QueueManager::Add(CircularData::Name::FILE);
QueueManager::Add(CircularData::Name::HELPER);

//Starting the threads
QueueManager::StartAudioThread();
QueueManager::StartFileThread();
QueueManager::StartHelperThread();
```

Thread Spinning



```
//THIS SHOULD BE SPINNING WHILE RECEIVING DATA FROM A CIRCULAR QUEUE
//THE DATA WILL INCLUDE ALL THE ACTIONS TO THE ASOUND OBJECT
while (!QuitFlag)
{
    if (!pIn_Queue->isEmpty())
    {
        CommandBase* command;

        if (pIn_Queue->PopFront(command)) //we found a command
        {
            Debug::out("Executing Command\n");
            command->Execute();
            delete command;
        }
    }

    TimerEventManager::Update(Time::quotient(threadTime.toc(), Time(TIME_ONE_MILLISECOND)));
}
```

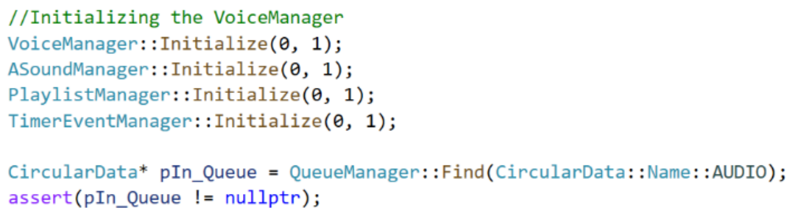


```
void QueueManager::StartAudioThread()
{
    QueueManager* _instance = privInstance();
    std::thread t_audio(Test_Main, std::ref(_instance->QuitFlag));
    Debug::SetName(t_audio, "Main Audio");
    t_audio.detach();
}
```

The threads spin until the exit flag is turned, this is done by executing the GDSoundEngine::StopEngine() method. In the meantime it is consuming the elements from the CircularQueue which is the Queue specified with QueueManager::Add(CircularData::Name::AUDIO), after being consumed they can be disposed of and so we call their destructors.

Example of Manager Setup

(Audio Thread)



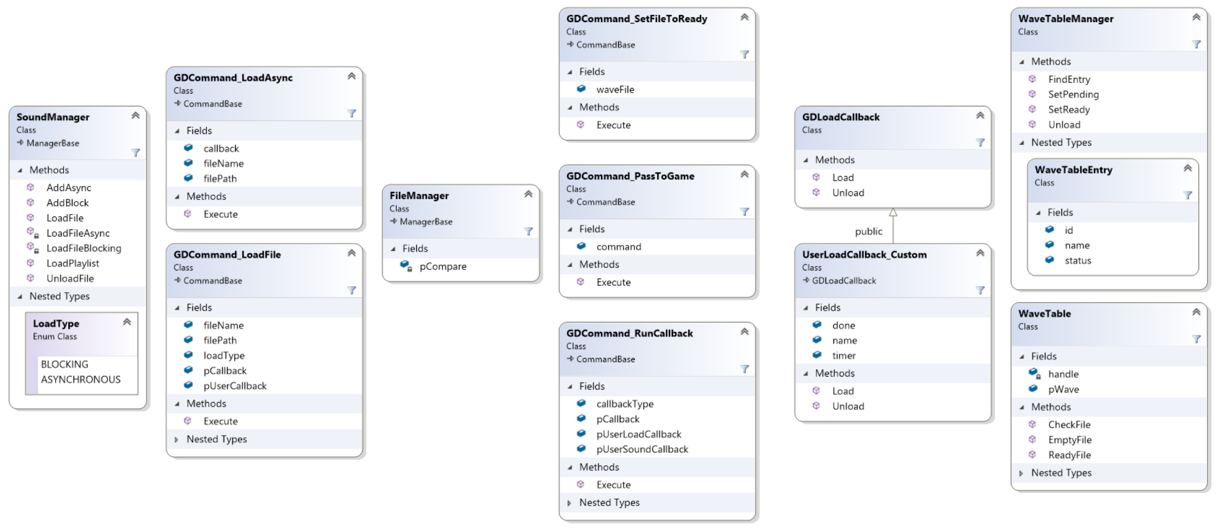
```
//Initializing the VoiceManager
VoiceManager::Initialize(0, 1);
ASoundManager::Initialize(0, 1);
PlaylistManager::Initialize(0, 1);
TimerEventManager::Initialize(0, 1);

CircularData* pIn_Queue = QueueManager::Find(CircularData::Name::AUDIO);
assert(pIn_Queue != nullptr);
```

Files, Voices and XAudio2

Before we can start playing our sound files, we first need to load them, GDSoundEngine takes care of that in two ways:

- Synchronous Load (Load Blocking) – This type of loading will block the main thread from continuing, this is useful in the case of preload, when you need to load assets before proceeding.
- Asynchronous Load (Load Async) – This type of loading will let the thread continue and will notify the user when it is finished loading by using a user created custom GDLoadCallback, useful for loading in the background.

The different load types differ in a small way but otherwise perform the same function, which is to create a File node in the File Manager and update the Wave Table from Pending to Ready.



This happens in 5 steps (in the case of Asynchronous Loading)

1. User calls the LoadFileAsync call on the SoundManager
    a. This checks on the game thread if the WaveID we want to load is already in the WaveTable with a Ready or Pending status, if it is then it just sends the UserCallback to the Helper thread which marshals it back to the Game thread, if it is not there then it sends the LoadAsync command to the audio thread.
    b. If it does not exist in the table, it sends the LoadAsync command to the Audio thread.

2. The LoadAsync just creates the LoadFile Command (This is used in both Blocking and Async except that in Async a UserLoadCallback is added) and sends it to the File thread to be processed
3. The LoadFile command Adds the WaveID to the WaveManager (FileManager in this case) and attaches the UserCallback in the call
4. The Add method creates 3 commands while also initializing the WaveFile and setting the entry as Pending in the WaveTable
    a. SetFileToReadyCommand which just Adjust the table entries with Ready status (this is sent to the Helper thread)

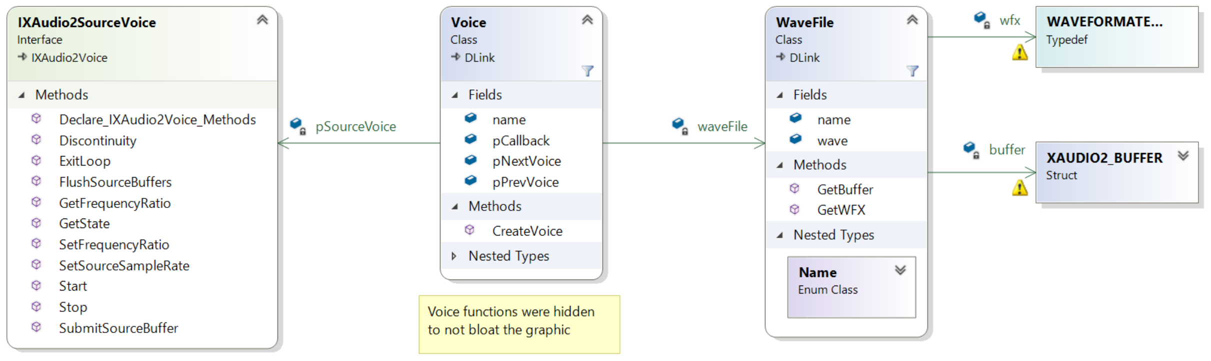
    b. PassToGameCommand this just tells the Helper thread to send it back to the Game thread for execution (this is needed to guarantee ordering, we don't want to set it to ready before Executing the callback) (this is sent to the Helper thread)
    c. RunCallbackCommand which executes any callback that it is passed to it (UserLoadCallback or InternalCallback) (this is passed as a parameter to the PassToGameCommand)

5. The UserCallback executes its LoadMethod in the Game thread and if the user wants to execute some specific code, then they can override the Load method since it is a virtual function on its derived class GDLoadCommand

Since the WaveTable is mutex protected it guarantees that only one thread can access it at a time.

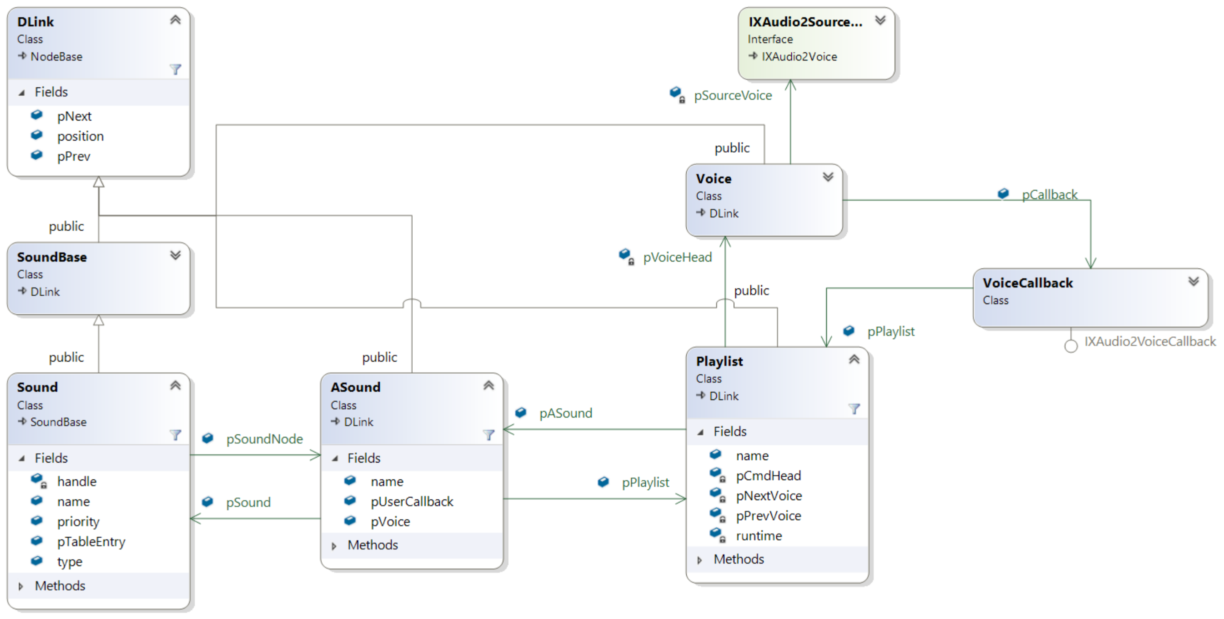
As for the Files (WaveFiles), these are objects that have a pointer to a WAVEFORMATEXTENSIBLE(wfx) as well as a pointer to an XAUDIO2_BUFFER(buffer), these values get populated by calling the GDAudio::LoadAudioFile method, it reads through the wave file's content and finds its *fourccFMT* chunk then loads it into the wfx, and then looks for the *fourccDATA* and loads that into the buffer.

This data lives in the FileManager, and when we need to use it we can create a Voice object, when creating this object we specify what File it is going to be attached to, this object will then create a IXAudio2SourceVoice which will consume the buffer and wfx from the WaveFile.



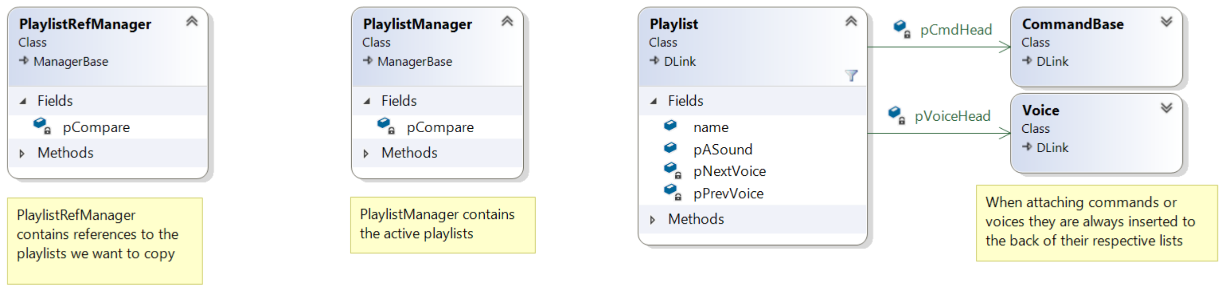
Sound and Playlists

Sound Creation looks like this:



When we create a Sound object, we also create its respective ASound object, this is an object that lives on the Audio thread side this makes it faster to access this object for any operation. Whenever we perform any operations on the Sound object it sends a command to the Audio thread and then accesses the ASound directly, it does not perform operations on the ASound from the Game thread, we want these functions separated to not cause thread issues.

As for playlists



The way that playlists work is that, given a playlist object with a particular name (SoundBase::Name) we can attach voices (Sound Files in WAV format) and attach commands, these voices will play one after the other. The minimum voices needed is one, while there is no limit for the maximum (so far).

LoadPlaylist(SoundBase::Name) this will create a playlist on the Playlist Reference Manager, whose purpose is to have the playlists readily available for consumption by other sound objects, whenever we need to create a sound with that playlists it just creates a copy of it, this is to allow instancing.

Creating the playlist on the PlaylistReferenceManager

```
//Load the playlist into the Reference manager
GDSoundEngine::LoadPlaylist(SoundBase::Sound_201);

//Attach voices to the playlist
GDSoundEngine::AttachVoice(SoundBase::Sound_201, WaveFile::Name::Intro_Mono);
GDSoundEngine::AttachVoice(SoundBase::Sound_201, WaveFile::Name::A_Mono);
GDSoundEngine::AttachVoice(SoundBase::Sound_201, WaveFile::Name::AtoB_Mono);
GDSoundEngine::AttachVoice(SoundBase::Sound_201, WaveFile::Name::B_Mono);
GDSoundEngine::AttachVoice(SoundBase::Sound_201, WaveFile::Name::BtoC_Mono);
GDSoundEngine::AttachVoice(SoundBase::Sound_201, WaveFile::Name::C_Mono);
GDSoundEngine::AttachVoice(SoundBase::Sound_201, WaveFile::Name::CtoA_Mono);
GDSoundEngine::AttachVoice(SoundBase::Sound_201, WaveFile::Name::End_Mono);

//Attach Voices to the playlist
GDSoundEngine::AttachCommand(SoundBase::Sound_201, 0, CommandBase::Play);
GDSoundEngine::AttachCommand(SoundBase::Sound_201, 0, CommandBase::SetVolume, 0.7f); //sets the volume to 80%
GDSoundEngine::AttachCommand(SoundBase::Sound_201, 10000, CommandBase::SetPan, 1.0f, CommandBase::REPEAT); //set panning to left or right and repeat
GDSoundEngine::AttachCommand(SoundBase::Sound_201, 20000, CommandBase::SetPan, -1.0f, CommandBase::REPEAT);
GDSoundEngine::AttachCommand(SoundBase::Sound_201, 30000, CommandBase::SetPan, 0.0f, CommandBase::REPEAT);
```

Attaching voices is as simple as specifying which SoundBase::Name we want to affect and the WaveFile::Name we want to add.

Attaching commands is more complicated, it requires the SoundBase::Name we want to add it to, the Time in millisecons, and the command we want to add. If the command requires a set value, like the SetVolume or SetPan command then the next value is a float value, the last value is a tag to specify if the command gets repeated, if so, it will retrigger itself in the same time interval specified.

In order to create a Sound object we first need to load the playlist (this will be replaced with a serialized object at a later point but for now we are creating the playlists ourselves) and then we simply call the GDSoundEngine::AddBlock or AddAsync and specify the same playlist name we used in the example above. This will copy the playlist from the Playlist Reference Manager into the PlaylistManager and will link it to the particular Sound object.

```
Sound* pA = GDSoundEngine::AddBlock(SoundBase::Sound_201);
pA->StartPlaylist(10);
```

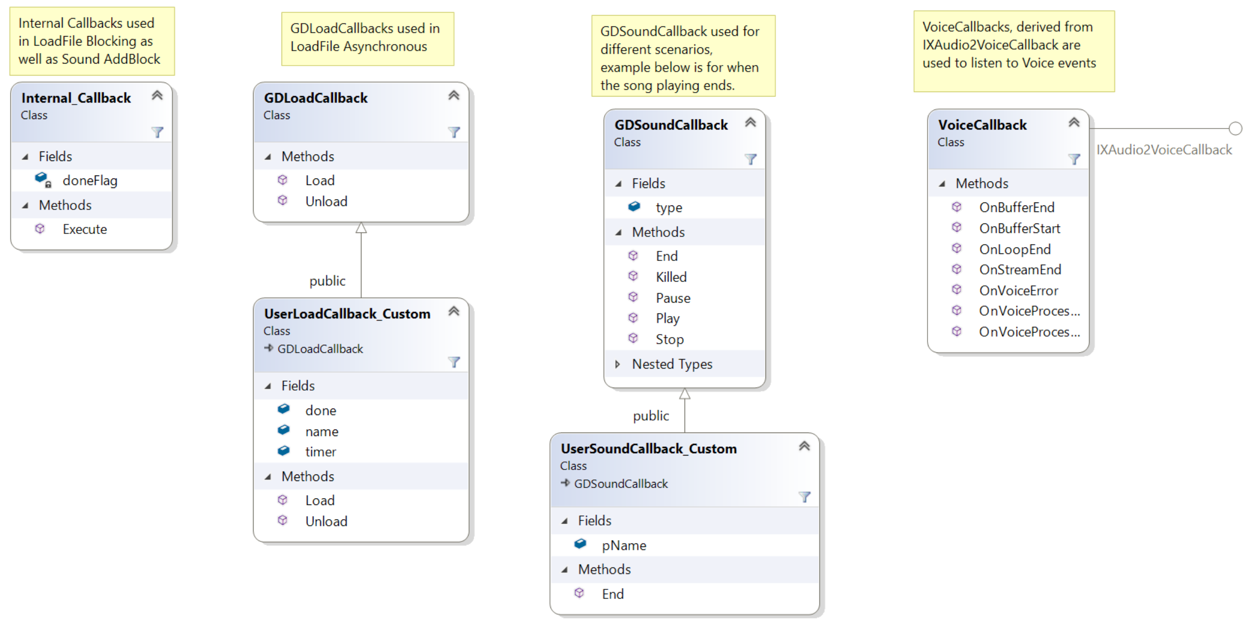
AddBlock will make the sound available immediately for use while AddAsync creates it while the thread continues, if you need to use the sounds later and not immediately then Async is recommended.

Calling the StartPlaylist method on the Sound object created will trigger the PlaylistStart command, this will send all the commands to the TimerEvent Manager to be executed. The int value specified is optional and it specifies the priority in which it will be played.

The way priority work is, the lower the number the higher the priority, if the PriorityTable is filled and the user tries to playlist then it will check if there is a value it can replace (one with lower priority, if there are lower priority entries then it will look for others with its same priority and replace the one that has been playing the longest). This system is to prevent too many sound effects to play at the same time since we set a limit on the number of sounds playable.

Callbacks

GDSoundEngine supports 4 different types of callbacks


Internal Callbacks used in LoadFile Blocking as well as Sound AddBlock

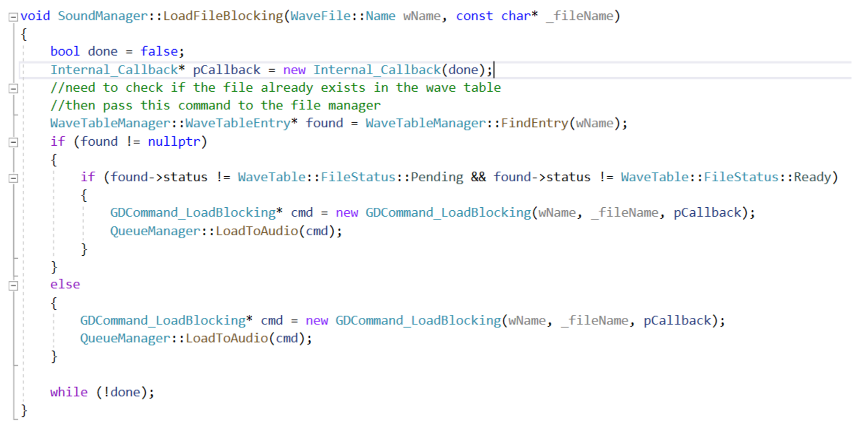
GDLoadCallbacks used in LoadFile Asynchronous

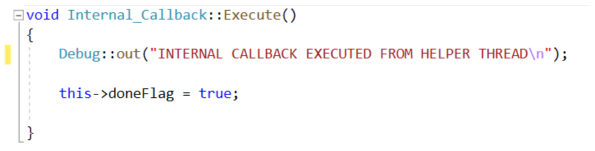
GDSoundCallback used for different scenarios, example below is for when the song playing ends.

VoiceCallbacks, derived from IXAudio2VoiceCallback are used to listen to Voice events

InternalCallbacks – These are callbacks used for Blocking loads, they simply turn the done flag true when the task is finished allowing the thread to continue working.

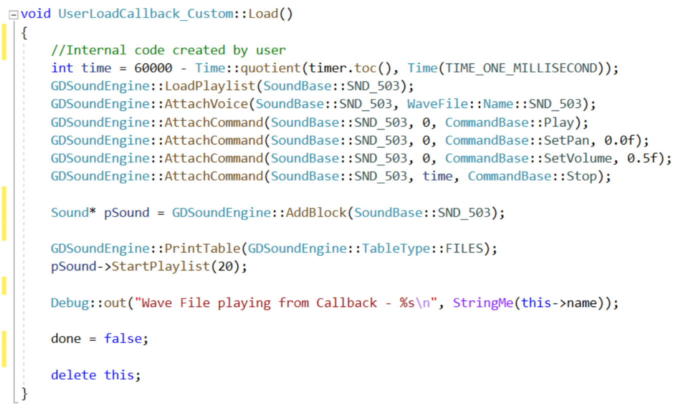
Here the process spins indefinitely, when it is finished the Internal_Callback's Execute function is called, turning the flag to true and exiting the loop.


```
void SoundManager::LoadFileBlocking(WaveFile::Name wName, const char* _fileName)
{
    bool done = false;
    Internal_Callback* pCallback = new Internal_Callback(done);
    //need to check if the file already exists in the wave table
    //then pass this command to the file manager
    WaveTableManager::WaveTableEntry* found = WaveTableManager::FindEntry(wName);
    if (found != nullptr)
    {
        if (found->status != WaveTable::FileStatus::Pending && found->status != WaveTable::FileStatus::Ready)
        {
            GDCommand_LoadBlocking* cmd = new GDCommand_LoadBlocking(wName, _fileName, pCallback);
            QueueManager::LoadToAudio(cmd);
        }
    }
    else
    {
        GDCommand_LoadBlocking* cmd = new GDCommand_LoadBlocking(wName, _fileName, pCallback);
        QueueManager::LoadToAudio(cmd);
    }

    while (!done);
}
```


```
void Internal_Callback::Execute()
{
    Debug::out("INTERNAL CALLBACK EXECUTED FROM HELPER THREAD\n");

    this->doneFlag = true;

}
```

GDLoadCallbacks – These are callbacks used for Asynchronous loads, these are specified by the user directly, by deriving from the GDLoadCallback we are using the strategy pattern, the engine will then run the Load function derived from GDLoadCallback and execute the user's code in the Game engine, this way, we ensure the threads are unaffected by the user's code.

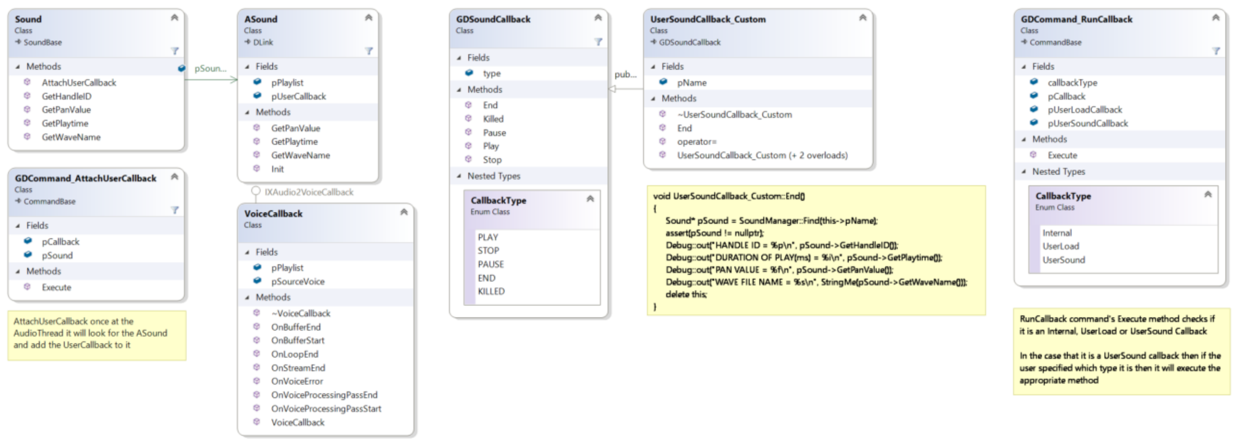
Example of user created code:



```cpp
void UserLoadCallback_Custom::Load()
{
    //Internal code created by user
    int time = 60000 - Time::quotient(timer.toc(), Time(TIME_ONE_MILLISECOND));
    GDSoundEngine::LoadPlaylist(SoundBase::SND_503);
    GDSoundEngine::AttachVoice(SoundBase::SND_503, WaveFile::Name::SND_503);
    GDSoundEngine::AttachCommand(SoundBase::SND_503, 0, CommandBase::Play);
    GDSoundEngine::AttachCommand(SoundBase::SND_503, 0, CommandBase::SetPan, 0.0f);
    GDSoundEngine::AttachCommand(SoundBase::SND_503, 0, CommandBase::SetVolume, 0.5f);
    GDSoundEngine::AttachCommand(SoundBase::SND_503, time, CommandBase::Stop);

    Sound* pSound = GDSoundEngine::AddBlock(SoundBase::SND_503);

    GDSoundEngine::PrintTable(GDSoundEngine::TableType::FILES);
    pSound->StartPlaylist(20);

    Debug::out("Wave File playing from Callback - %s\n", StringMe(this->name));

    done = false;

    delete this;
}
```

Note: The Files, Voices and XAudio2 section shows the full flow of Asynchronous loading.

As for GDSoundCallbacks



AttachUserCallback once at the AudioThread it will look for the ASound and add the UserCallback to it

```cpp
void UserSoundCallback_Custom::End()
{
    Sound* pSound = SoundManager::Find(this->pName);
    assert(pSound != nullptr);
    Debug::out("HANDLE ID = %p\n", pSound->GetHandleID());
    Debug::out("DURATION OF PLAY(ms) = %i\n", pSound->GetPlaytime());
    Debug::out("PAN VALUE = %f\n", pSound->GetPanValue());
    Debug::out("WAVE FILE NAME = %s\n", StringMe(pSound->GetWaveName()));
    delete this;
}
```

RunCallback command's Execute method checks if it is an Internal, UserLoad or UserSound Callback

In the case that it is a UserSound callback then if the user specified which type it is then it will execute the appropriate method

When we create a Sound object (using GDSoundManager's AddBlock or AddAsync) we can attach a user callback to it, this will trigger a AttachUserCallback command and pass that onto the pASound of the respective Sound object. With this callback attached, when the XAudio2's OnStreamEnd method is called it will create the RunCallback method if there is a user callback attached and will send that callback to the Game thread to be executed.

The execution depends on what the user implemented, the custom callbacks are derived from GDSoundCallback and they can override the method they want to execute (they are set to virtual optional) and by setting the type they want it to be the callback's method is executed (this is functionality I would like to upgrade in the future). In the example above since the UserSoundCallback_Custom implements the End method and it is CallbackType::END then when the

RunCallback command is executed it will look for that type and call the End method, and since this command is sent to the Game thread, the user will see it directly.

Lastly VoiceCallbacks

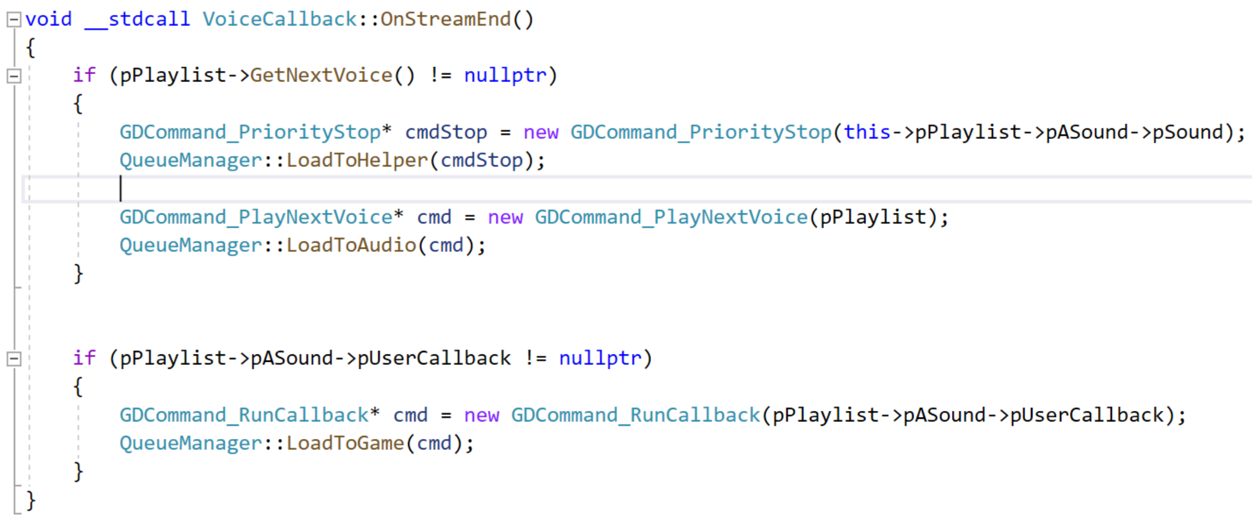
VoiceCallbacks are derived from IXAudio2VoiceCallback and they simply notify us when certain events occur in the XAudio2 thread, the same way we implement our own callbacks.

It will notify us when the following actions have occurred:

- OnBufferStart – When the source voice is about to process a new buffer data.
- OnBufferEnd – When the source voice is finished processing the buffer data.
- OnStreamEnd – When the source voice has finished playing.
- OnVoiceProcessingPassEnd – When the processing pass for a voice is finished.
- OnVoiceProcessingPassStart – When the processing pass is about to start, this is used for optimal feeding off data before starvation.
- OnLoopEnd – When the source voice loop has finished.
- OnVoiceError – When there is a critical error while processing the voice.

These methods are triggered on the XAudio2 thread, so in our case, to free that thread from any major work we delegate anything it needs to do to other threads, and in the case of UserSoundCallbacks we can send those callbacks directly to the Game thread to be processed.

In the sample used above, if the sound playing ends, it will check if there is another voice in the playlist and play it (this is functionality I would like to change in the future to make it so that I can assign the callback myself if I need to) and if there is a custom callback added it will create a RunCallback command and send it to the Game thread to be executed.

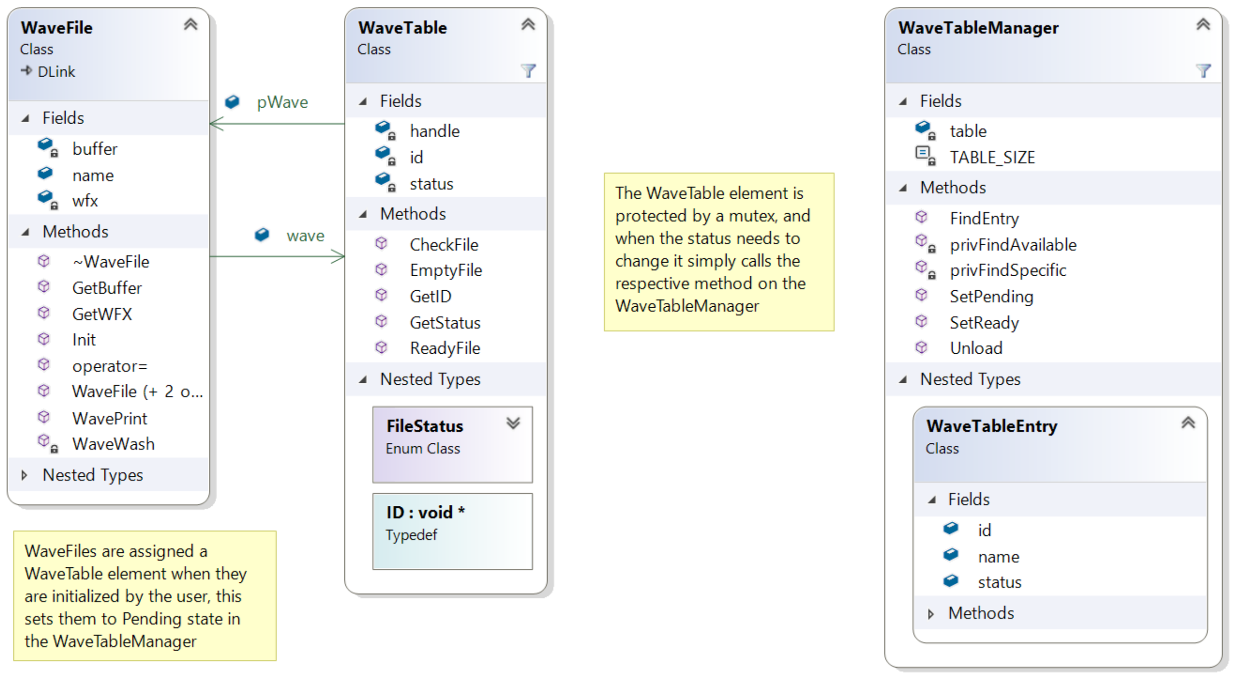


```
void __stdcall VoiceCallback::OnStreamEnd()
{
    if (pPlaylist->GetNextVoice() != nullptr)
    {
        GDCommand_PriorityStop* cmdStop = new GDCommand_PriorityStop(this->pPlaylist->pASound->pSound);
        QueueManager::LoadToHelper(cmdStop);

        GDCommand_PlayNextVoice* cmd = new GDCommand_PlayNextVoice(pPlaylist);
        QueueManager::LoadToAudio(cmd);
    }


    if (pPlaylist->pASound->pUserCallback != nullptr)
    {
        GDCommand_RunCallback* cmd = new GDCommand_RunCallback(pPlaylist->pASound->pUserCallback);
        QueueManager::LoadToGame(cmd);
    }
}
```

WaveTable and PriorityTable

GDSoundEngine uses two different of tables in its lifecycle, the WaveTable or FileTable and the PriorityTable.

WaveTable consist of keeping a record of all the active Files available and their status, they could be Pending if the command to load has been sent but it has not finished, and Ready when the file is stored in the FileManager.

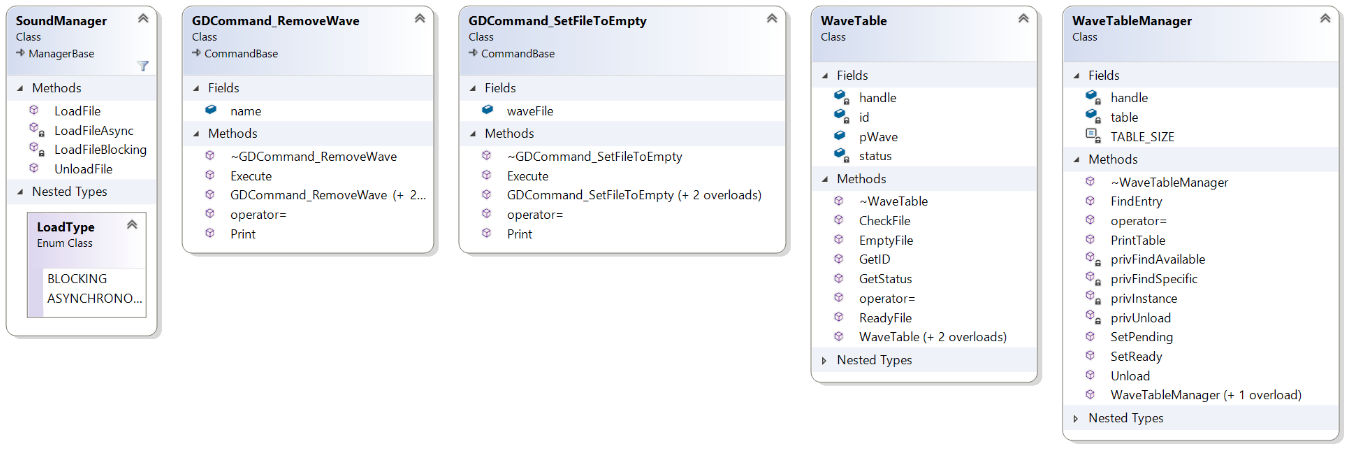


The WaveTable element is protected by a mutex, and when the status needs to change it simply calls the respective method on the WaveTableManager

WaveFiles are assigned a WaveTable element when they are initialized by the user, this sets them to Pending state in the WaveTableManager

Note: The full flow of the WaveTable update is shown in the Files, Voices and XAudio2 section

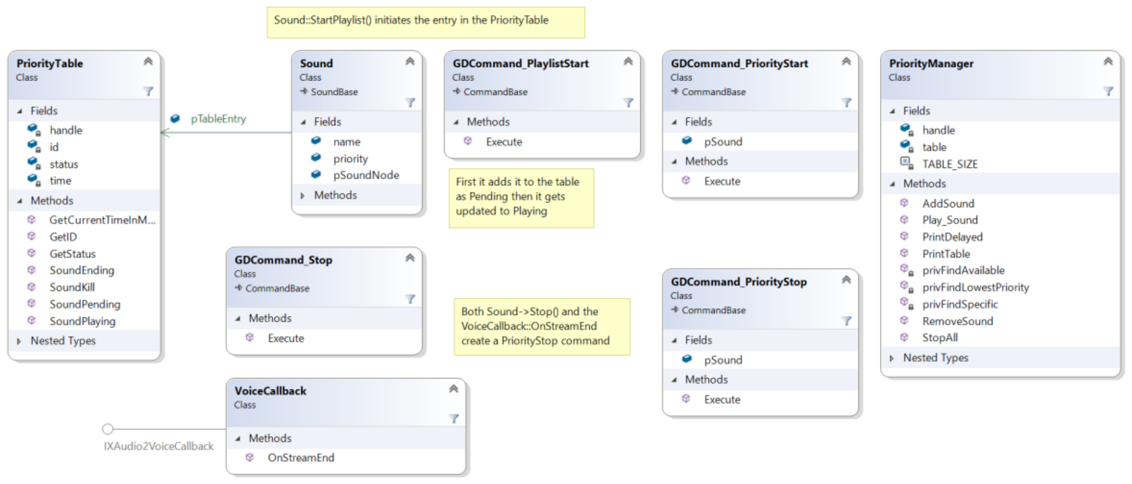
When the user decides to remove a file from the system then they can do so by calling the UnloadFile method.



When the user calls the UnloadFile and specifies the WaveID to remove, it creates a RemoveWaveCommand and sends it to Audio thread since this is the thread that handles the WaveFiles (not their loading but their usage), The RemoveWaveCommand just checks that the WaveID returns a valid WaveFile and creates a SetFileToEmptyCommand which sends to the Helper thread, this command will update the table. At the same time the RemoveWaveCommand is removing the WaveFile from the WaveManager (FileManager in this case).

The SetFileToEmpty just calls the EmptyFile method on the WaveFile's WaveTable entry.
I decided to go with this approach rather than go through the WaveTableManager because the Table is not mutex protected but the WaveTable entry is. The EmptyFile method just calls the privUnload method on the WaveTableManager which sets its id to nullptr, its status to Empty and its Name to Uninitialized.

As for the PriorityTable



Say your player is in a scene, it's a sunny park and there are lots of people talking, birds chirping, cars driving around, weather effects, etc… It might be overwhelming to the player to have to listen to all these sound effects at once, and so the priority table helps us cull sound effects based on their priority. The lower the number the higher the priority.

When triggering Sound::StartPlaylist, the sound object creates a PriorityStart command, which is sent to the Helper/Aux thread. This is to check that the entry can be added into the table first. If there is no space for the entry and there is no element to replace then it will not trigger the PlaylistStart command. Thus, the sound is never played.

This PriorityStart command simply calls the PriorityManager::Play_Sound(Sound*) method which handles the table update.

To add it to the table it needs to do the following:

- Check for an empty space, if it finds it then it updates that entry with the details of the sound we are about to play then triggers the PlaylistStart command
- If there is no empty space, check for the lowest priority and replace that, then triggers the PlaylistStart command
- If there is no empty space, and there is no priority lower than the new one, check for priorities of the same level, and replace the one with the highest playtime.
- If it does not find an entry with a higher priority than itself then it skips the play

When the new sound replaces another, it stops that sound first, sets its internals to Killed and then replaces that entry with the new one and plays.

The Priority Manager is protected by a handle/mutex system, the Play_Sound method has a mutex, so it locks the access, adds the sound to the table then plays it and releases the access. The StopAll and RemoveSound methods have this mutex as well since they are updating the table.

Printing

The GDSoundEngine has 2 methods for printing, PrintTable and Print Managers, PrintTable will Print the contents of the specified table

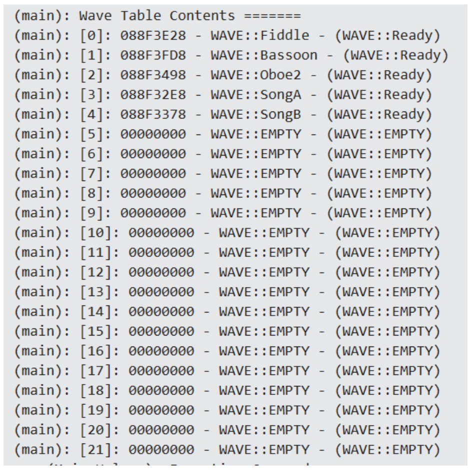
As part of PrintTable there are two options PrintTable and PrintTableDelayed

```
GDSoundEngine::PrintTable(GDSoundEngine::TableType::FILES);
GDSoundEngine::PrintTableDelayed(GDSoundEngine::TableType::PRIORITY);
```

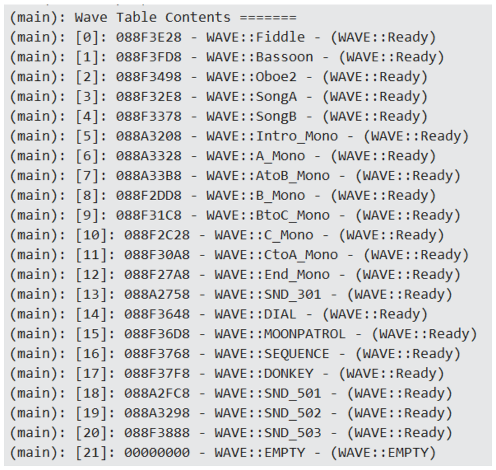
PrintTable will print out the contents on the Output window immediately, while PrintTableDelayed will send a command to the Helper thread and have it printed, this ensures that the result is printed after any other instruction sent to the Helper thread (ensures ordering), you can tell it is being printed at the Helper thread due to the since our printing method is thread aware.

Example of resutls

FILES (Index, Pointer, Name, Status)

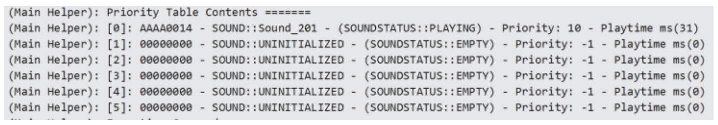


```
(main): Wave Table Contents =======
(main): [0]: 088F3E28 - WAVE::Fiddle - (WAVE::Ready)
(main): [1]: 088F3FD8 - WAVE::Bassoon - (WAVE::Ready)
(main): [2]: 088F3498 - WAVE::Oboe2 - (WAVE::Ready)
(main): [3]: 088F32E8 - WAVE::SongA - (WAVE::Ready)
(main): [4]: 088F3378 - WAVE::SongB - (WAVE::Ready)
(main): [5]: 00000000 - WAVE::EMPTY - (WAVE::EMPTY)
(main): [6]: 00000000 - WAVE::EMPTY - (WAVE::EMPTY)
(main): [7]: 00000000 - WAVE::EMPTY - (WAVE::EMPTY)
(main): [8]: 00000000 - WAVE::EMPTY - (WAVE::EMPTY)
(main): [9]: 00000000 - WAVE::EMPTY - (WAVE::EMPTY)
(main): [10]: 00000000 - WAVE::EMPTY - (WAVE::EMPTY)
(main): [11]: 00000000 - WAVE::EMPTY - (WAVE::EMPTY)
(main): [12]: 00000000 - WAVE::EMPTY - (WAVE::EMPTY)
(main): [13]: 00000000 - WAVE::EMPTY - (WAVE::EMPTY)
(main): [14]: 00000000 - WAVE::EMPTY - (WAVE::EMPTY)
(main): [15]: 00000000 - WAVE::EMPTY - (WAVE::EMPTY)
(main): [16]: 00000000 - WAVE::EMPTY - (WAVE::EMPTY)
(main): [17]: 00000000 - WAVE::EMPTY - (WAVE::EMPTY)
(main): [18]: 00000000 - WAVE::EMPTY - (WAVE::EMPTY)
(main): [19]: 00000000 - WAVE::EMPTY - (WAVE::EMPTY)
(main): [20]: 00000000 - WAVE::EMPTY - (WAVE::EMPTY)
(main): [21]: 00000000 - WAVE::EMPTY - (WAVE::EMPTY)
```

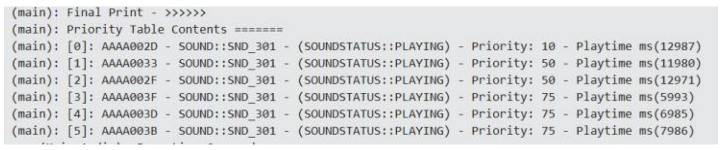


```
(main): Wave Table Contents =======
(main): [0]: 088F3E28 - WAVE::Fiddle - (WAVE::Ready)
(main): [1]: 088F3FD8 - WAVE::Bassoon - (WAVE::Ready)
(main): [2]: 088F3498 - WAVE::Oboe2 - (WAVE::Ready)
(main): [3]: 088F32E8 - WAVE::SongA - (WAVE::Ready)
(main): [4]: 088F3378 - WAVE::SongB - (WAVE::Ready)
(main): [5]: 088A3208 - WAVE::Intro_Mono - (WAVE::Ready)
(main): [6]: 088A3328 - WAVE::A_Mono - (WAVE::Ready)
(main): [7]: 088A33B8 - WAVE::AtoB_Mono - (WAVE::Ready)
(main): [8]: 088F2DD8 - WAVE::B_Mono - (WAVE::Ready)
(main): [9]: 088F31C8 - WAVE::BtoC_Mono - (WAVE::Ready)
(main): [10]: 088F2C28 - WAVE::C_Mono - (WAVE::Ready)
(main): [11]: 088F30A8 - WAVE::CtoA_Mono - (WAVE::Ready)
(main): [12]: 088F27A8 - WAVE::End_Mono - (WAVE::Ready)
(main): [13]: 088A2758 - WAVE::SND_301 - (WAVE::Ready)
(main): [14]: 088F3648 - WAVE::DIAL - (WAVE::Ready)
(main): [15]: 088F36D8 - WAVE::MOONPATROL - (WAVE::Ready)
(main): [16]: 088F3768 - WAVE::SEQUENCE - (WAVE::Ready)
(main): [17]: 088F37F8 - WAVE::DONKEY - (WAVE::Ready)
(main): [18]: 088A2FC8 - WAVE::SND_501 - (WAVE::Ready)
(main): [19]: 088A3298 - WAVE::SND_502 - (WAVE::Ready)
(main): [20]: 088F3888 - WAVE::SND_503 - (WAVE::Ready)
(main): [21]: 00000000 - WAVE::EMPTY - (WAVE::EMPTY)
```

PRIORITY (Index, HandleID, Name, Status, Priority, Playtime)



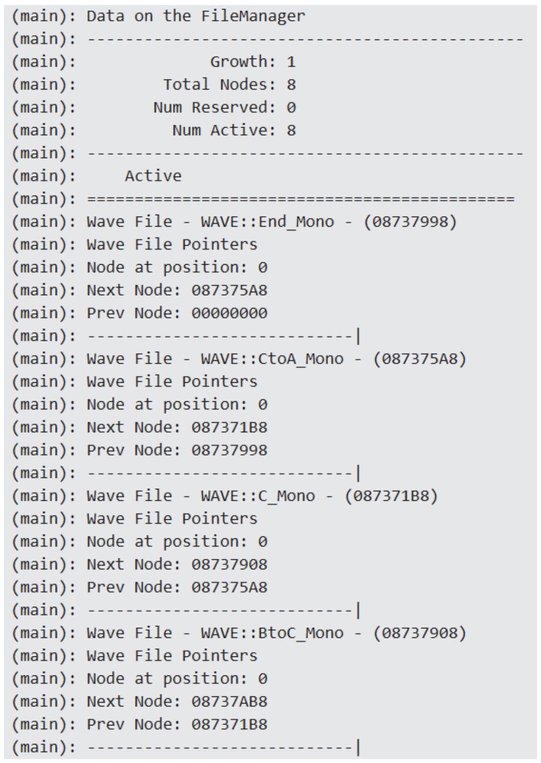
```
(Main Helper): Priority Table Contents =======
(Main Helper): [0]: AAAA0014 - SOUND::Sound_201 - (SOUNDSTATUS::PLAYING) - Priority: 10 - Playtime ms(31)
(Main Helper): [1]: 00000000 - SOUND::UNINITIALIZED - (SOUNDSTATUS::EMPTY) - Priority: -1 - Playtime ms(0)
(Main Helper): [2]: 00000000 - SOUND::UNINITIALIZED - (SOUNDSTATUS::EMPTY) - Priority: -1 - Playtime ms(0)
(Main Helper): [3]: 00000000 - SOUND::UNINITIALIZED - (SOUNDSTATUS::EMPTY) - Priority: -1 - Playtime ms(0)
(Main Helper): [4]: 00000000 - SOUND::UNINITIALIZED - (SOUNDSTATUS::EMPTY) - Priority: -1 - Playtime ms(0)
(Main Helper): [5]: 00000000 - SOUND::UNINITIALIZED - (SOUNDSTATUS::EMPTY) - Priority: -1 - Playtime ms(0)
```



```
(main): Final Print - >>>>>>
(main): Priority Table Contents =======
(main): [0]: AAAA002D - SOUND::SND_301 - (SOUNDSTATUS::PLAYING) - Priority: 10 - Playtime ms(12987)
(main): [1]: AAAA0033 - SOUND::SND_301 - (SOUNDSTATUS::PLAYING) - Priority: 50 - Playtime ms(11980)
(main): [2]: AAAA002F - SOUND::SND_301 - (SOUNDSTATUS::PLAYING) - Priority: 50 - Playtime ms(12971)
(main): [3]: AAAA003F - SOUND::SND_301 - (SOUNDSTATUS::PLAYING) - Priority: 75 - Playtime ms(5993)
(main): [4]: AAAA003D - SOUND::SND_301 - (SOUNDSTATUS::PLAYING) - Priority: 75 - Playtime ms(6985)
(main): [5]: AAAA003B - SOUND::SND_301 - (SOUNDSTATUS::PLAYING) - Priority: 75 - Playtime ms(7986)
```

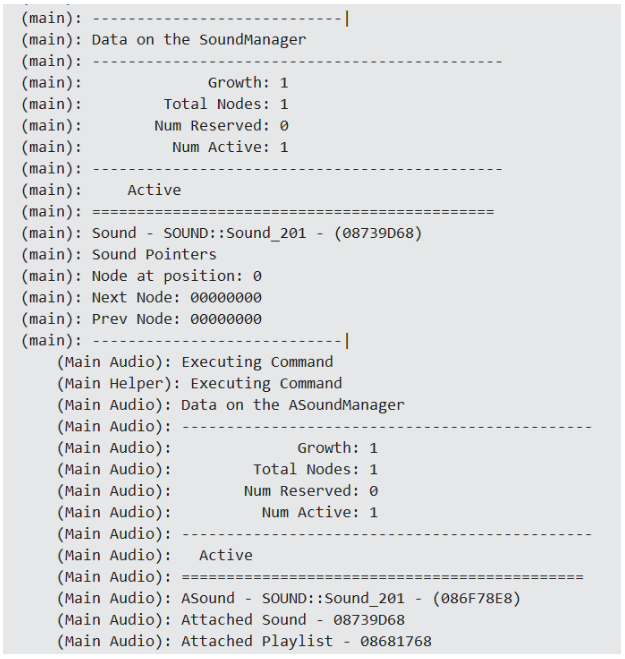
As for the PrintManager, the FILES tag specifies to Print out the File Manager and the SOUND tag specifies to print out the Sound and the ASound manager, these will also print out the contents of the playlist.

FILE



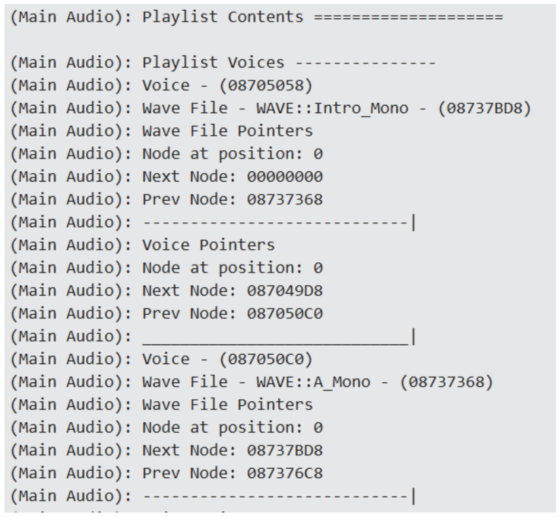
```
(main): Data on the FileManager
(main): ----------------------------------------------
(main):             Growth: 1
(main):        Total Nodes: 8
(main):       Num Reserved: 0
(main):         Num Active: 8
(main): ----------------------------------------------
(main):     Active
(main): ===========================================
(main): Wave File - WAVE::End_Mono - (08737998)
(main): Wave File Pointers
(main): Node at position: 0
(main): Next Node: 087375A8
(main): Prev Node: 00000000
(main): ---------------------------|
(main): Wave File - WAVE::CtoA_Mono - (087375A8)
(main): Wave File Pointers
(main): Node at position: 0
(main): Next Node: 087371B8
(main): Prev Node: 08737998
(main): ---------------------------|
(main): Wave File - WAVE::C_Mono - (087371B8)
(main): Wave File Pointers
(main): Node at position: 0
(main): Next Node: 08737908
(main): Prev Node: 087375A8
(main): ---------------------------|
(main): Wave File - WAVE::BtoC_Mono - (08737908)
(main): Wave File Pointers
(main): Node at position: 0
(main): Next Node: 08737AB8
(main): Prev Node: 087371B8
(main): ---------------------------|
```

SOUND

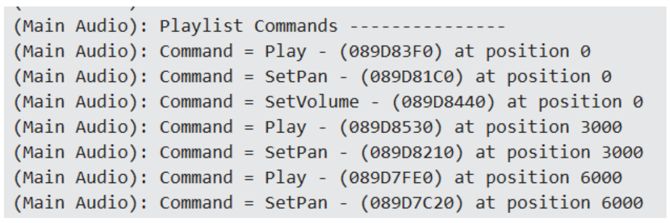


```
(main): ---------------------------|
(main): Data on the SoundManager
(main): ----------------------------------------------
(main):             Growth: 1
(main):        Total Nodes: 1
(main):       Num Reserved: 0
(main):         Num Active: 1
(main): ----------------------------------------------
(main):     Active
(main): ===========================================
(main): Sound - SOUND::Sound_201 - (08739D68)
(main): Sound Pointers
(main): Node at position: 0
(main): Next Node: 00000000
(main): Prev Node: 00000000
(main): ---------------------------|
    (Main Audio): Executing Command
    (Main Helper): Executing Command
    (Main Audio): Data on the ASoundManager
    (Main Audio): ----------------------------------------------
    (Main Audio):             Growth: 1
    (Main Audio):        Total Nodes: 1
    (Main Audio):       Num Reserved: 0
    (Main Audio):         Num Active: 1
    (Main Audio): ----------------------------------------------
    (Main Audio):   Active
    (Main Audio): ===========================================
    (Main Audio): ASound - SOUND::Sound_201 - (086F78E8)
    (Main Audio): Attached Sound - 08739D68
    (Main Audio): Attached Playlist - 08681768
```

These are printed as part of SoundManager Print as well



```
(Main Audio): Playlist Contents ====================

(Main Audio): Playlist Voices ---------------
(Main Audio): Voice - (08705058)
(Main Audio): Wave File - WAVE::Intro_Mono - (08737BD8)
(Main Audio): Wave File Pointers
(Main Audio): Node at position: 0
(Main Audio): Next Node: 00000000
(Main Audio): Prev Node: 08737368
(Main Audio): ---------------------------|
(Main Audio): Voice Pointers
(Main Audio): Node at position: 0
(Main Audio): Next Node: 087049D8
(Main Audio): Prev Node: 087050C0
(Main Audio): ___________________________|
(Main Audio): Voice - (087050C0)
(Main Audio): Wave File - WAVE::A_Mono - (08737368)
(Main Audio): Wave File Pointers
(Main Audio): Node at position: 0
(Main Audio): Next Node: 08737BD8
(Main Audio): Prev Node: 087376C8
(Main Audio): ---------------------------|
```

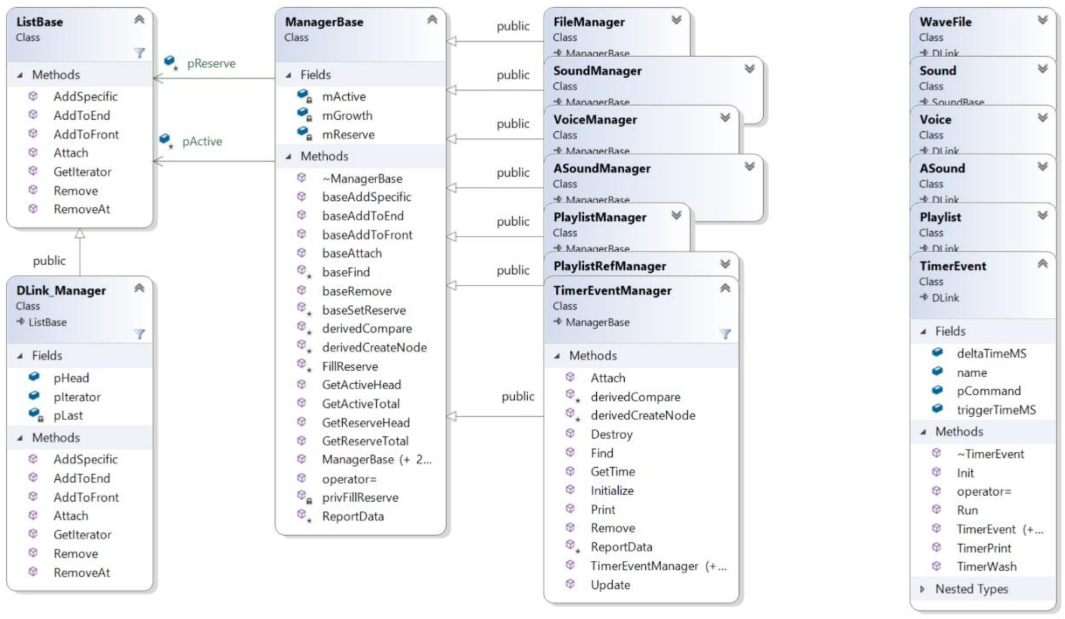


```
(Main Audio): Playlist Commands ---------------
(Main Audio): Command = Play - (089D83F0) at position 0
(Main Audio): Command = SetPan - (089D81C0) at position 0
(Main Audio): Command = SetVolume - (089D8440) at position 0
(Main Audio): Command = Play - (089D8530) at position 3000
(Main Audio): Command = SetPan - (089D8210) at position 3000
(Main Audio): Command = Play - (089D7FE0) at position 6000
(Main Audio): Command = SetPan - (089D7C20) at position 6000
```

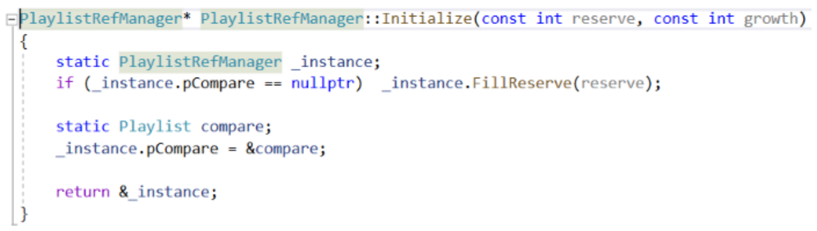
Design patterns used for this project

Singleton – (Managers are singleton instances)

UML Diagram



This is achieved using Scott Meyer's Singleton pattern


```
PlaylistRefManager* PlaylistRefManager::Initialize(const int reserve, const int growth)
{
    static PlaylistRefManager _instance;
    if (_instance.pCompare == nullptr)  _instance.FillReserve(reserve);

    static Playlist compare;
    _instance.pCompare = &compare;

    return &_instance;
}
```

General

The basic idea of the Singleton pattern is to streamline the use of a class to a single instance, this is used with all the Managers in this project.

The main thing to note is for singletons their constructors are private so if we want to use this pattern the user needs to call the Initialize method, which will create an instance of the class and store it within itself, if the user tries to call the Initialize method again then we check if there is an instance already created, if there is, we ignore this request (or show error if we needed to), however by making sure that all crucial calls to the manager call the initialize function we guarantee that the user will not use the Manager without creating an instance.

The Problem

The problem that we face with this pattern is that what if we had multiple scenes and each scene required an independent version of this singleton? Well in that case we would have to make some adjustments to our singleton, we have to make our constructor not private and then we add another instance of itself. Whenever we need to use this for a particular scene, we just create a separate instance of the class and set it as active while we are in that scene, if we switch scenes, we switch the active instance.

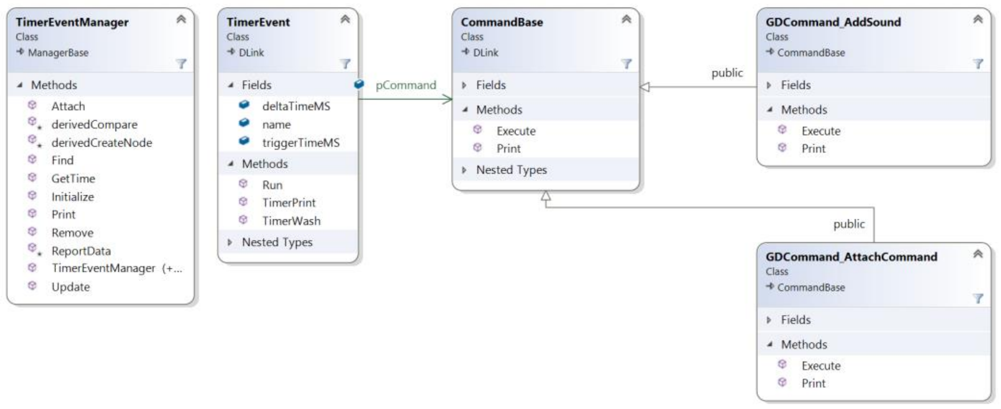
The benefit of this pattern

It is handy to keep track of only one instance of the class rather than having to create a variable for one and using that as a global variable, everything we need from the class is already static, and so all the methods are available to us.

This makes it so that everyone has access to the same instance rather than everyone creating their own.

Command with Timer Event Manager

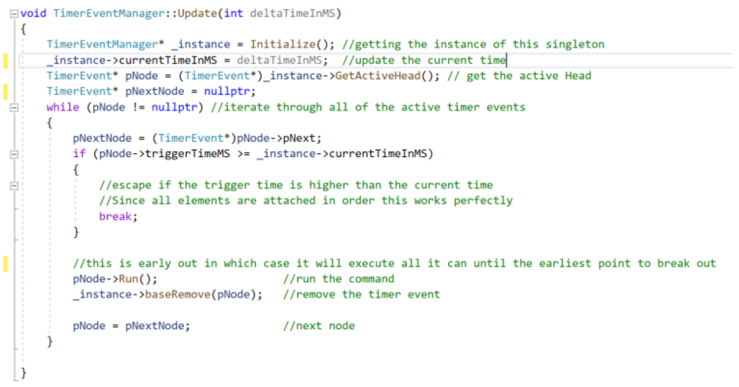
UML Diagram



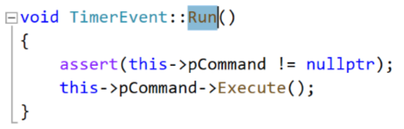
General

Commands are objects that execute their functions at a given time, we use them in conjunction with our Timer Events.

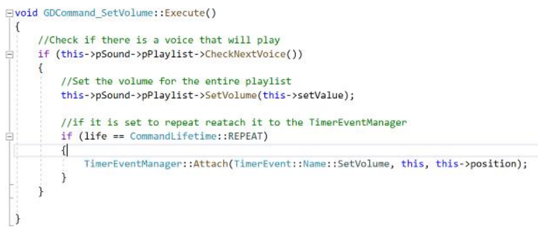
The design consists of an abstract Command class with an Execute method, then all our different Commands can use it to execute their functions as part of our Timer Events.



```
void TimerEventManager::Update(int deltaTimeInMS)
{
    TimerEventManager* _instance = Initialize(); //getting the instance of this singleton
    _instance->currentTimeInMS = deltaTimeInMS;  //update the current time
    TimerEvent* pNode = (TimerEvent*)_instance->GetActiveHead(); // get the active Head
    TimerEvent* pNextNode = nullptr;
    while (pNode != nullptr) //iterate through all of the active timer events
    {
        pNextNode = (TimerEvent*)pNode->pNext;
        if (pNode->triggerTimeMS >= _instance->currentTimeInMS)
        {
            //escape if the trigger time is higher than the current time
            //Since all elements are attached in order this works perfectly
            break;
        }

        //this is early out in which case it will execute all it can until the earliest point to break out
        pNode->Run();                   //run the command
        _instance->baseRemove(pNode);   //remove the timer event

        pNode = pNextNode;              //next node
    }

}
```



```
void TimerEvent::Run()
{
    assert(this->pCommand != nullptr);
    this->pCommand->Execute();
}
```



```
void GDCommand_SetVolume::Execute()
{
    //Check if there is a voice that will play
    if (this->pSound->pPlaylist->CheckNextVoice())
    {
        //Set the volume for the entire playlist
        this->pSound->pPlaylist->SetVolume(this->setValue);

        //if it is set to repeat reatach it to the TimerEventManager
        if (life == CommandLifetime::REPEAT)
        {
            TimerEventManager::Attach(TimerEvent::Name::SetVolume, this, this->position);
        }
    }

}
```

The Problem

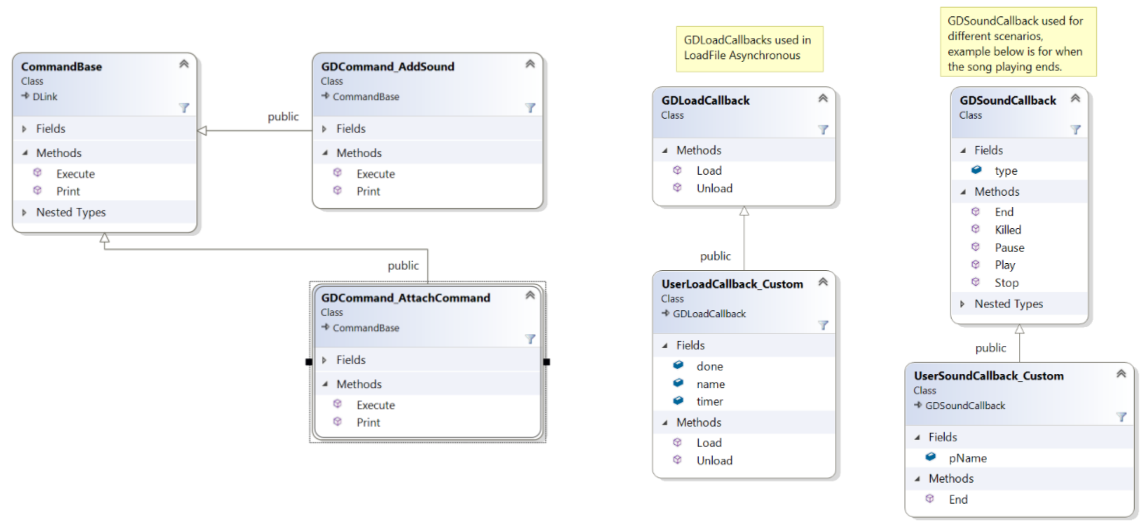
The main problem with these types of behavioral patterns is that if we don’t necessarily know what gets executed as all the command types have an Execute method.

The benefits of this pattern

This is the working force behind our engine, everything is set on timers and commands, this is how we are sending the instructions from one thread to the next.

Strategy – (Commands, Custom Callbacks fit this pattern)

UML Diagram



General

The main idea is to define algorithms related by type (in this case our Commands fit that pattern as well) and we just extrapolate all the main functions into abstract classes in the main class, and all it is all taken care of by its subclasses.

This pattern is very similar to the state pattern but in the case of state pattern we just switch the state objects, and the functionality is taken care of, but here we are pretty much subclassing with different takes on the abstract methods depending on the use.

The Problem

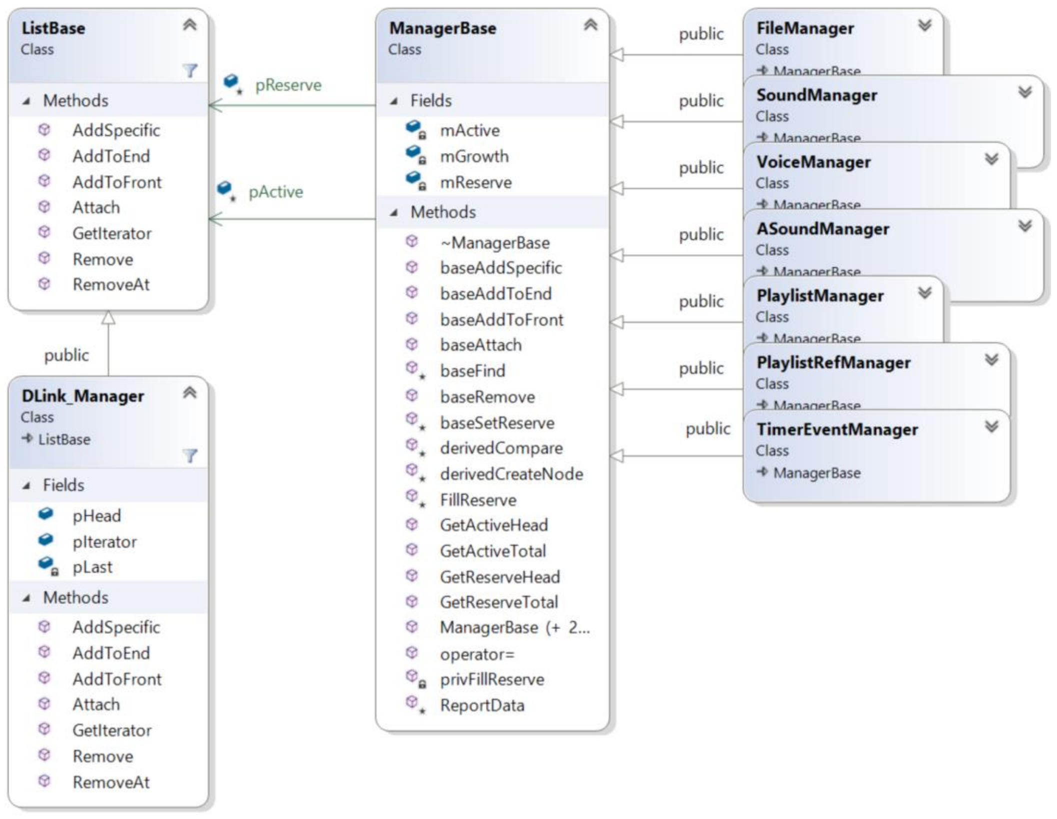
To change the behavior of a class you would need to create a different object.

The benefits of this pattern

Our commands implement this pattern to some degree, they are all children of a base class and the way its abstract methods are created means that all the subclasses can use these methods with their own twist. In the case of our Custom Callbacks, regardless of their implementation they are derived from the GDSoundCallback and GDLoadCallback, and as such they need to implement certain functions. The user must provide one and if they decide to just not do so then the default behavior will trigger which is just an empty function, if they implement their own version then that will be executed instead.

Object Pooling – (Managers and Double Linked Lists)

UML Diagram



General

This pattern consists of creating two separate pools, one active and one reserve, these pools work together tightly, whenever we create any objects we just take from the reserve and add it to active then we can initialize that element as we see fit, whenever we are done with the elements we can remove them by adding them back to the reserve list and washing them of any data, then they get recycled again when new objects are needed.

It will also grow in size depending on how many elements are needed, if there are not enough elements in the reserve list then it will simply add more and use those.

The Problem

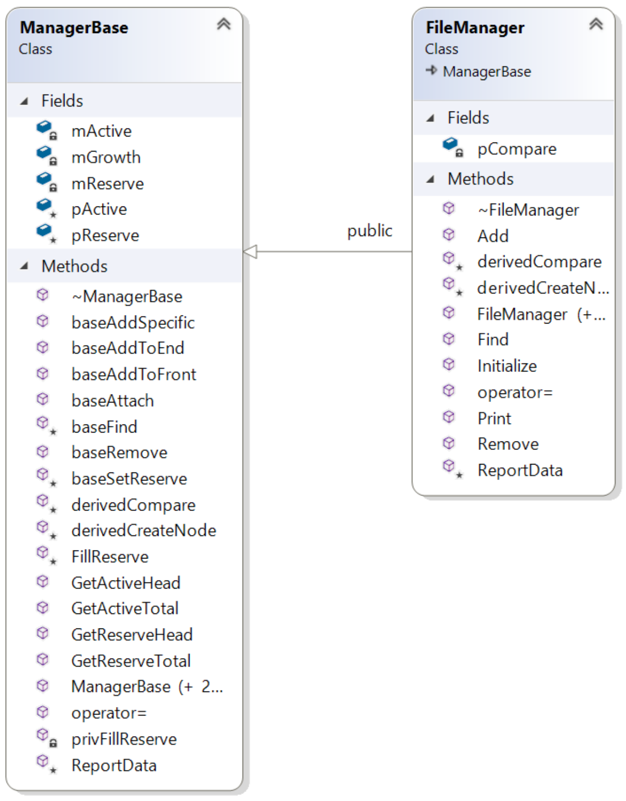
This pattern brings one problem, and it is that no object ever gets destroyed so no memory is freed, it's all kept in reserve pools.

The Benefit in this case

The files and playlists are always kept in the manager whenever the user needs them, no need to re add them.

Template Pattern – (For Managers and their derived implementations)

UML Diagram



General

The Template design pattern allows for a parent class to have some methods implemented and the child classes could use these derived methods and implement their own version based on their needs.

The Problem

The need for this pattern arises at the fact that if we had multiple classes with the same functions except one or two steps then we could just have a base class have the basic skeleton of the process and the child classes can use them in their own way.

The benefits of this pattern

Once we turn all our methods into steps then we can extrapolate each and use what we need then the rest could be on the child classes.

Post-Mortem

Without a doubt this was one of the most technical projects I have ever worked on, I think working on the Final Fantasy Demo beats it since it had more moving parts, this was more concentrated on one task, "Play a Sound and Play it Properly". I had to do quite a bit of research and still am, in which case this document may undergo some updates.

Good documentation is undervalued, having documentation that is well written and detailed makes the process or wrapping an existing systems to suit your own needs easier. It was while going through the process of wrapping the Win32's File system that it became an arduous task just to keep up with it and it was a bit confusing, same thing happened while wrapping the XAudio2 system to create this audio engine. I believe that there is merit to writing a documentation that is technical, as you cannot really create a document that solves the user's problem. But you can guide the user to the possibilities once they get passed that first hurdle.

The ability to wrap an already existing class into your own version of it should not be taken lightly either, like using different connectors for an appliance but you don't have the necessary cable to wire them, you use an adapter to finish the connection. This is basically what we are doing, instead of taking the complicated base "vanilla" code we are simply creating an adapter we can later use to simplify our development. Had we not had these handy getting to where we are now would've been extremely difficult to follow.

Large systems like the graphics system need to be disassembled before we work on them, if we want to work on it, we will want to know a few things:

- Can its features be separated into small components?
- Can we reuse these components and how viable it would be to refactor these to suit our needs?
- Given that we are working on it on a per week basis, would the core functionality need more refactoring in the future?

For the audio engine our task was simple, we need to associate a file in wav format to a voice that will use XAudio2 to consume its buffer and play. Something as simple as that led to this system. Wrapping XAudio2, Win32's File system, creating Managers to hold our data, Queues to spin our threads, then delegating instructions to each. That also meant that since we have threads, we have to protect ourselves, and so mutex was introduced.

When we were first introduced to the sound engine concept we were pretty much put in the shoes of a new engineer, no experience whatsoever and we were given an API to understand bit by bit and use it. The way that I handled this was basically to read the code carefully, take my time with it, then work on top of it, if there were things, I could use in a different way would take some liberties in modifying it.

Regardless, I do thank DePaul for giving me the opportunity to work on this project and inspiring me to continue its development even after class had finished. Which led to this document. And thank you reader for taking your time to go through my experience in creating the GDSoundEngine (First true pass of it)