Concordia University – SOEN6471

# Linux Multimedia Studio

## Milestone 3

| | |
|---|---|
| *Alexandre Hudon* | *9580433* |
| *Simon Symeonidis* | *5887887* |
| *Sardar Maninder Singh* | *6555519* |
| *Pavleen Kaur* | *9754733* |
| *Amir Hakim* | *4050711* |

*Due: March 3, 11 marks*

Presented to: Dr. Peter Rigby

# Linux Multimedia Studio

## 1. Part 1: Conceptual versus Actual Diagrams

### 1.1. Github

As requested, the assignment code, and documents are all located here:
https://github.com/AHudon/SOEN6471_LMMS

### 1.2. Summary of Project

LMMS (Linux Multimedia Studio) is a DAW (Digital Audio Workstation), very similar to commercial ones, such as FL Studio. It provides facilities for you to engineer synthesizers in order to create instruments, and also piano rolls to create possible melodies. There exists a beat-bassline interface as well, to assign repetitive patterns quickly.

### 1.3. Conceptual Diagram

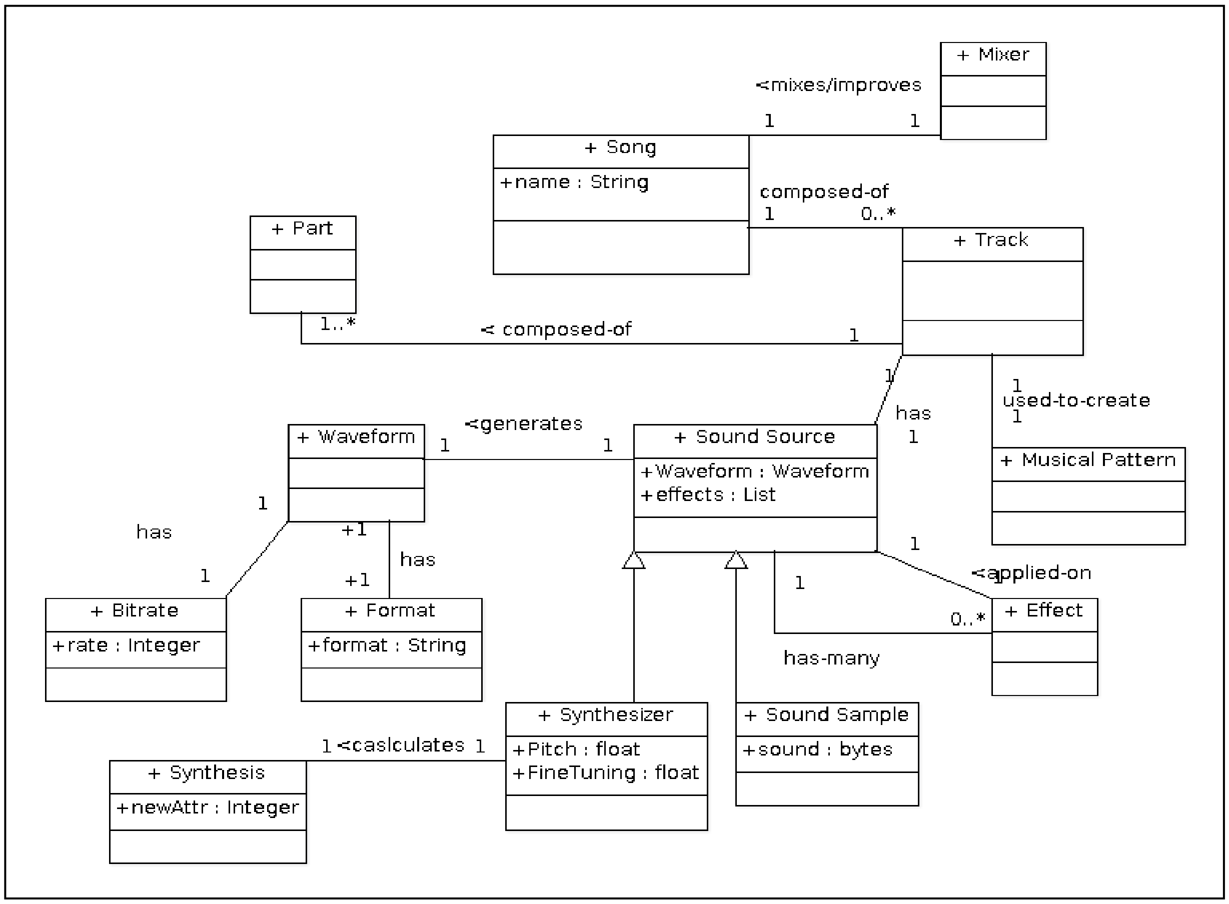


**Figure 1**: General overview of the LMMS system from Milestone 2.

We have taken the time to make modifications according to the feedback received from Milestone #2 prior to comparing our conceptual diagram with the actual diagram.

## 1.4. Class Diagrams of Actual System

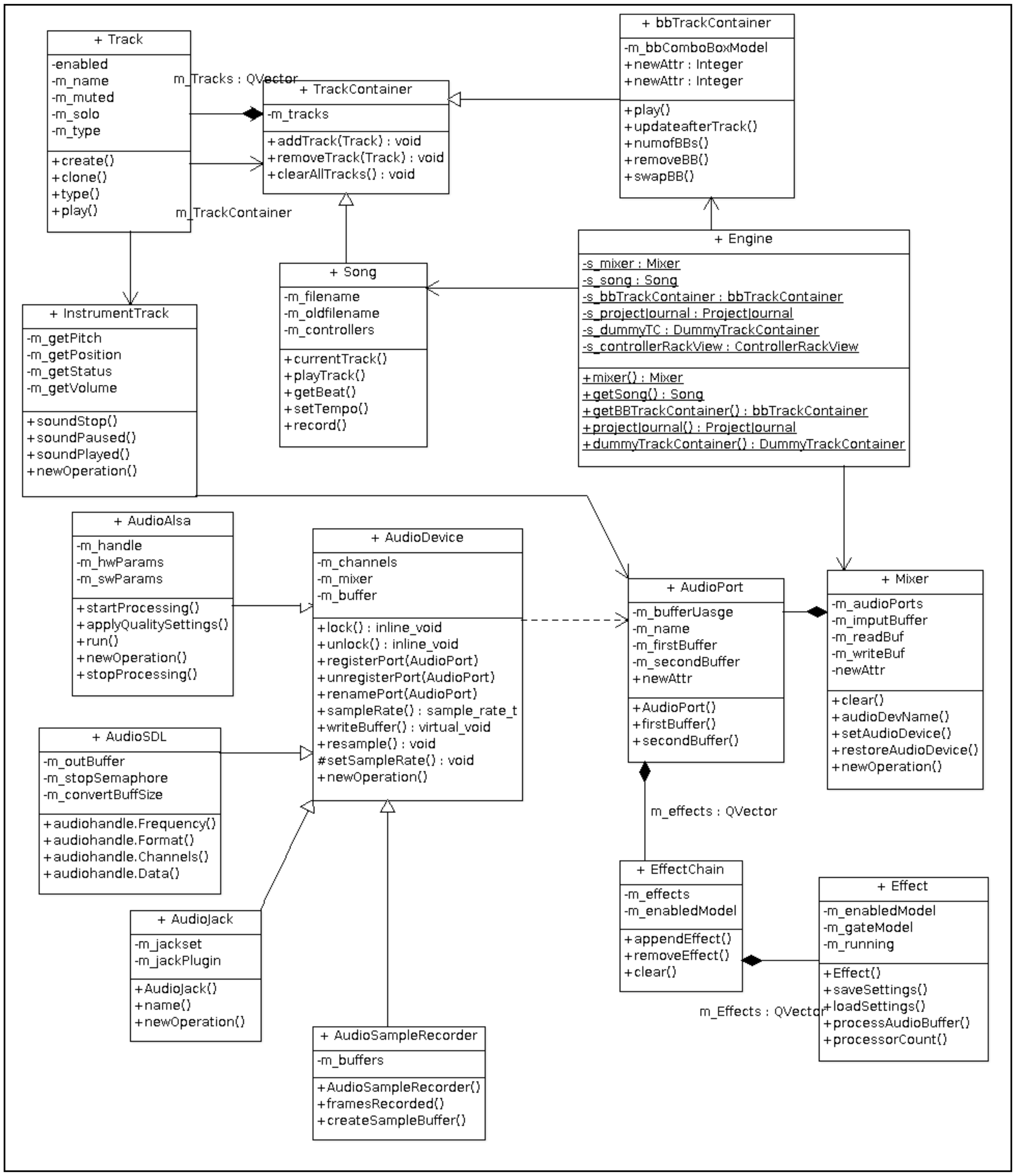


**Figure 2**: Zooming on Track

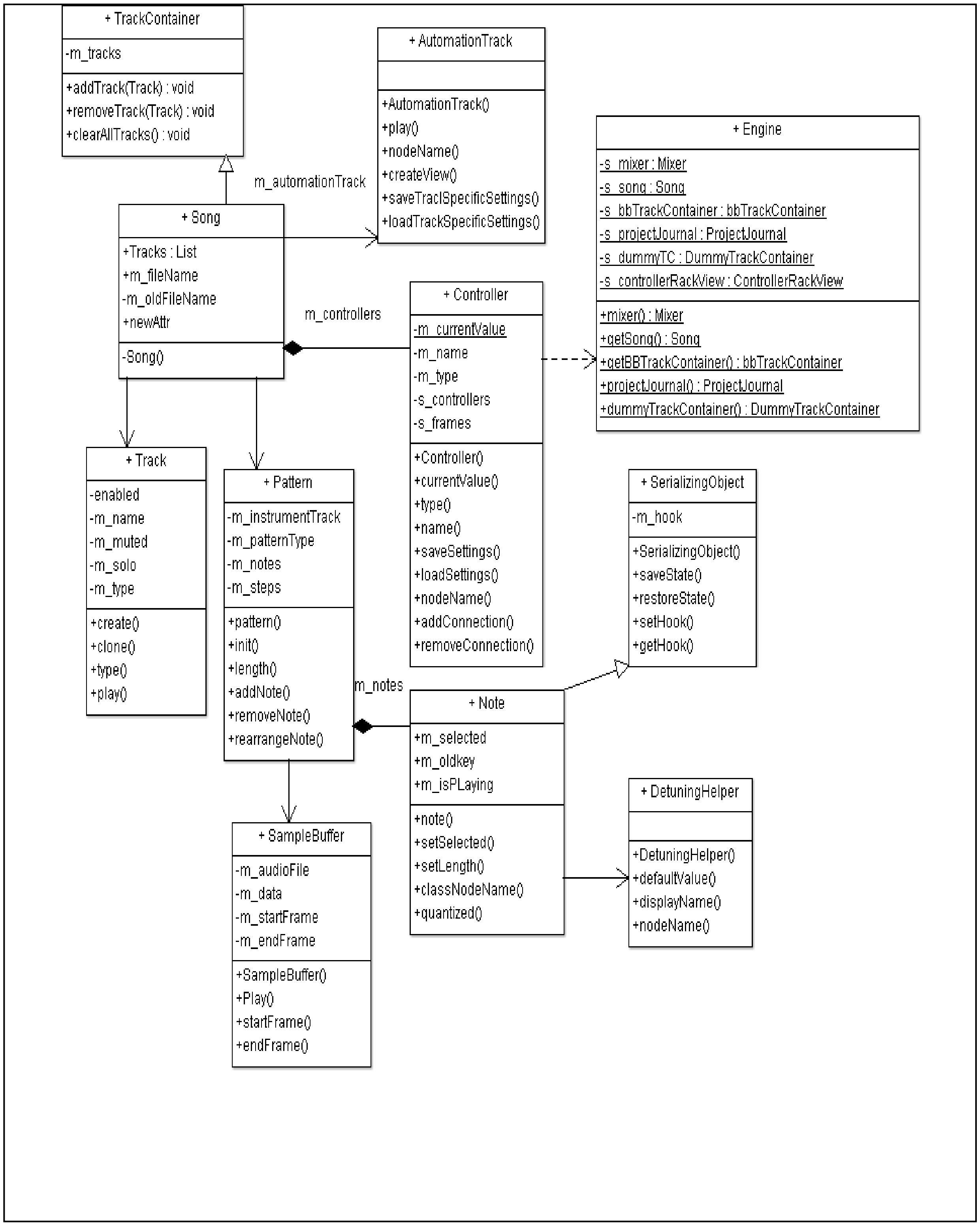


**Figure 3**: Zooming on Song

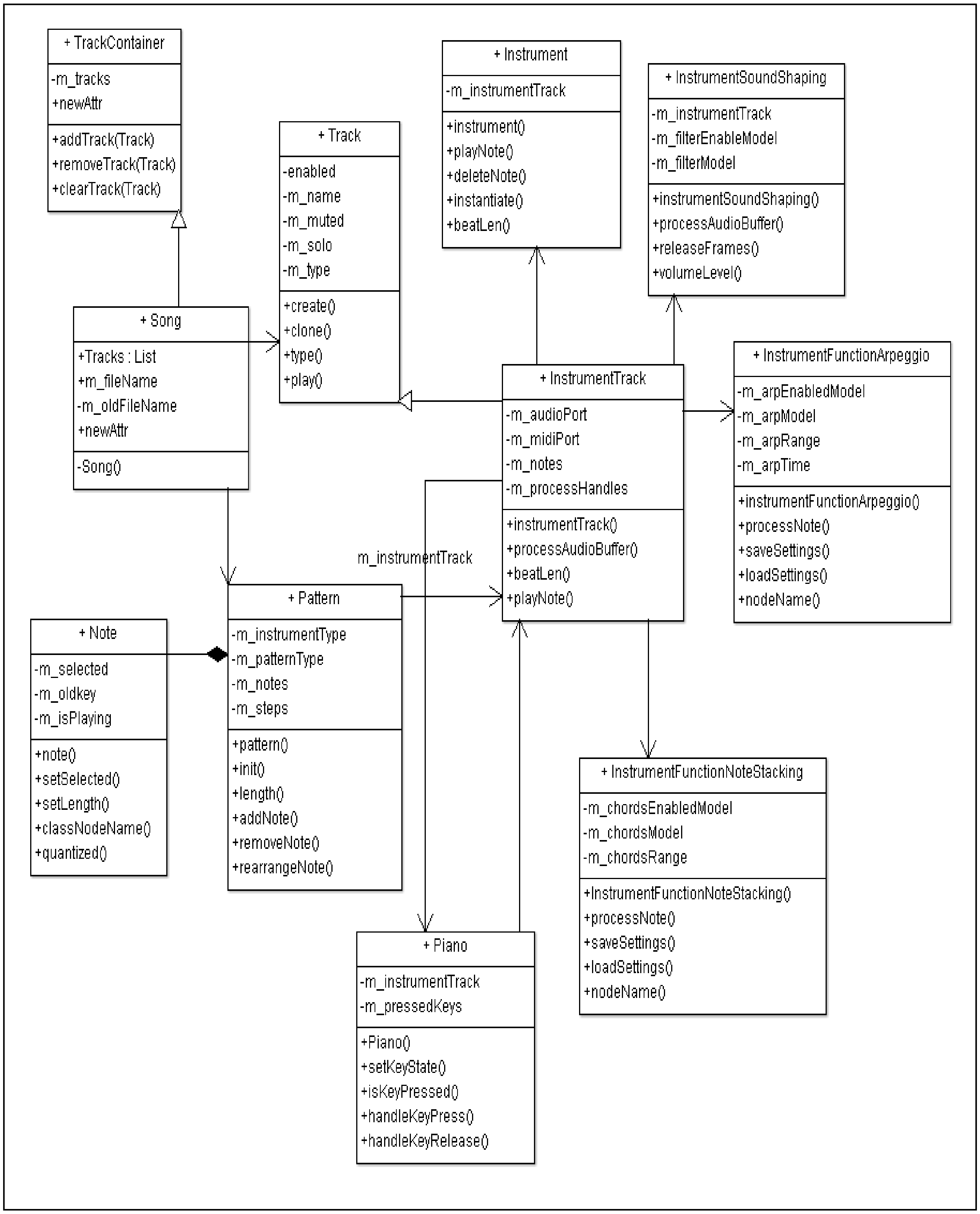


**Figure 4**: Zooming on Instrument

## 1.5 Discrepancies/Similarities: Conceptual Classes to Actual Classes

There is a discrepancy between the concepts and the actual classes. For complex systems, such as the Linux Multimedia Studio, this will be much more obvious, because complex systems often require auxiliary classes to match concepts together. For example aggregations might almost be similar, but mechanisms introduced (e.g. Lazy Loading) are highly likely to break any possible 1-to-1 relationships /mapping from the Domain Model to the actual Class Diagrams.

**Mixer, Song, Track and Musical Pattern**

These entities map coherently (however not a 1-1 relationship) to the actual classes of the system (**Mixer – MixerSong – Song**, **Track-Track**), with the exception that Musical Pattern is not exactly the same as bbTrackContainer.cpp. The bbTrackContainer is a container for other tracks, and tracks aggregate notes.

When we specified Musical Pattern as an entity of the domain model, we meant a collection of notes that are repeated in different measures (musical measures). This is implemented using some classes that are not listed in our class diagram (namely Pattern, and trackContentObject which together form a simple mechanism that makes this work) because we observed that other parts of the system required much more care in terms of refactoring and quality improvement.

On the subject of Song (from the actual class diagram, Figure 3), we rapidly spot a problem: inheritance abuse. In human thinking it would be obvious that a song, in this case, is a collection of 'stacked' tracks. However in the case of the software, namely in Figure 2, we see that Song inherits from the TrackContainer. This is something that we will look into, as a more proper way to couple these classes is by association.

For class Mixer, the domain entity is much different than in the class diagram. The Mixer in the domain model was directly linked to the song, whereas in the practical case, there exists an Engine class that acts as a mediator between said Mixer, and Song.

The Engine almost acts as a singleton due to its global state: a collection of static fields usable from any part in the system, as observed in the source code (adding all direct relations would make the diagram very messy).

Track, Figure 2, is much more decomposed in the software than in our domain model. In the domain model we thought that a Track entity would suffice (all track related behavior and attributes would belong to Track). However in the code, we found out that it requires a track container (TrackContainer) that aggregates Track instances.

Another unexpected issue we found is that Song directly inherits by this container. Track also aggregates contents using a 'TCO' object as observed in the code, less cryptically meaning "Track Content Object". This class is extended in order to provide different Track Content types *inside* a track. For example, a repeated pattern, versus a collection of notes that are only played once.

**Bitrate and Format**

The Bitrate and Format were not in the class diagram. Though they make sense in a domain model (for example a person might like high quality (bitrate) mp3s (format) or some other person might like high quality Ogg Vorbis (format)), they're taken care of in a different fashion in practice - mainly delegated as fields to an exporter plugin (eg: LAME for mp3 etc), and is the case in the system.

**Synthesis, Synthesizer and Sound Sample**

They are taken care of in a much different fashion than what we had in mind. These are not included in the diagram. The rationale is that they seem to have a proper mechanism to deal with this already hence no refactoring needed, and we foresaw that it would be too complex in this course to look into it further (on Linux distributions, Wine (Windows Emulator) is used along in order to provide functionality for integration of VST/ instruments, and this can potentially make things complicated and irrelevant to the course material).

Overall the way that the above is achieved, is by having a Model store a generic plugin. The plugin can be anything: a synthesizer, a sample, a VST instrument, etc. The PluginView contains this information, and it is used with any other possible window (recall that models are coupled with UI in our case, so this hackery is possible).

The InstrumentTrack , Figure 4, contains a PluginView, and some other generic parts: Instrument{SoundShapingView, FunctionNoteStackingView, FunctionArpeggioView, InstrumentMidiIOView}, EffectRackView, which are present for any synthesizer, and affect the sound output in different ways.

SoundShaping is for decay, sustain, and other volume related alterations to the sound output. Note stacking is actually for chords. Arpeggio lets you choose a scale and an octave range to arpeggiate upon. The midi IO view is to interface with external real, musical keyboards. The EffectRackView is the container for possible effect plugins.

## 1.6 Necessary Evils

This is not a part of the Milestone 3 specifications, but we thought it was important to discuss.

Some of the domain objects in the application are coupled with the Qt (UI) library. In principle we know this is not favorable. In this specific case, it doesn't matter. There has been a lot of work done on the user interface, and it is *very* specific to the user base [ref: 1- 4].

Also, Qt one of the most popular option for developers to use in contrast with other technologies such as GTK+, wxWidgets, Fox Toolkit, etc. New user interface toolkits rarely appear, and regardless change is unlikely to happen. If there was a richer collection of user interface libraries to use, then this might have been a concern.

### 1.7 Reverse Engineering Tool (Doxygen)

We used Doxygen as a reverse engineering tool. If a system using Doxygen has the extra package 'graphviz', then apart from extracting the API of the application, we are able to extract many other diagrams of interest, namely class diagrams, and collaboration diagrams.

However in order to present the work, ArgoUML was used in order to make diagrams that are more readable. As previously stated, some classes in those diagrams were left blank or partly filled because in this project, long classes are quite common. This already raises an obvious need for refactoring.

### 1.8 Relationship between 2 classes

For the two classes requested, we chose *TrackContainer*, and *Track*. We show how the relation exists, in this case the aggregation. We also show the similar methods and attributes that are demonstrated in the class diagram by selecting to paste *specific* code. The definition, and implementation of these classes is quite long, so most code is omitted.

Track Definition (header) [/include/track.h]

```
class EXPORT track : public Model, public JournallingObject
{
  ...
private:
	TrackContainer* m_trackContainer;
	TrackTypes m_type;
	QString m_name;
	int m_height;

	BoolModel m_mutedModel;
	BoolModel m_soloModel;
```

```
        bool m_mutedBeforeSolo;

        bool m_simpleSerializingMode;

        tcoVector m_trackContentObjects;

        friend class trackView;
```

Track Implementation (cpp) [/src/core/track.h]

```
track * track::create( TrackTypes _tt, TrackContainer * _tc )

{
        track * t = NULL;

        switch( _tt )

        {

                case InstrumentTrack: t = new ::InstrumentTrack( _tc ); break;

                case BBTrack: t = new bbTrack( _tc ); break;

                case SampleTrack: t = new ::SampleTrack( _tc ); break;

                case AutomationTrack: t = new ::AutomationTrack( _tc ); break;

                case HiddenAutomationTrack:

                        t = new ::AutomationTrack( _tc, true ); break;

                default: break;

        }

        _tc->updateAfterTrackAdd();


        return t;

}

void track::clone()

{

        QDomDocument doc;

        QDomElement parent = doc.createElement( "clone" );

        saveState( doc, parent );

        create( parent.firstChild().toElement(), m_trackContainer );

}

// pure virtual functions (located in header)
```

```
TrackTypes type() const
{
		return m_type;
}
// to be implemented by track base classes
virtual bool play( const MidiTime & _start, const fpp_t _frames,
  const f_cnt_t _frame_base, int _tco_num = -1 ) = 0;
```

TrackContainer (header) [/include/TrackContainer.h]

```
class EXPORT TrackContainer : public Model, public JournallingObject

{
  Q_OBJECT

public:

  typedef QVector<track *> TrackList;

  TrackContainer();

  virtual ~TrackContainer();

  ...

private:

  TrackList m_tracks;

  friend class TrackContainerView;

  friend class track;

 ...

}
```

TrackContainer Implementation (cpp) [/src/core/track.h]

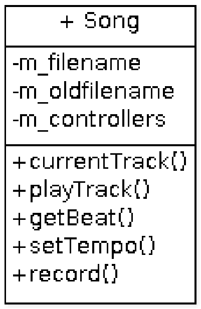
```
void TrackContainer::addTrack( track * _track )
{
	if( _track->type() != track::HiddenAutomationTrack )
	{
		m_tracksMutex.lockForWrite();
		m_tracks.push_back( _track );
		m_tracksMutex.unlock();
		emit trackAdded( _track );
	}
}
void TrackContainer::removeTrack( track * _track )
{
	int index = m_tracks.indexOf( _track );
	if( index != -1 )
	{
		m_tracksMutex.lockForWrite();
		m_tracks.remove( index );
		m_tracksMutex.unlock();

		if( engine::getSong() )
		{
			engine::getSong()->setModified();
		}
	}
}
void TrackContainer::clearAllTracks()
{
	//m_tracksMutex.lockForWrite();
	while( !m_tracks.isEmpty() )
	{
		delete m_tracks.first();
	}
	//m_tracksMutex.unlock();
}
```

## 2 . Code Smells and System Level Refactorings

Re-engineering the actual class diagrams for the Linux Multimedia Studio project helped us in identifying code smells that are impacting the maintainability of the software. We decided to highlight major code smells that require system-level refactoring. Other smaller refactoring are listed in Annex A.

**[1]Large Class and Large Method**



Class Song (song.cpp) is discussed below:

This class does not only contains the elements (attributes) that are pertinent to a song but also contains all the necessary artifacts to render a song, which makes this class unnecessarily large and one of its method (processNextBuffer(), line 191) contains over 200 lines of code and a maelstrom of nested switch and if-else-if statements.

To increase cohesion (song should not be responsible itself for the rendering of songs), we adopted the *Replace Type Code with State Strategy* [5]refactoring.

**Steps to fix the Large Method Code Smell**

1. playMode is the argument being passed in the switch statement (line 201) of processNextBuffer(). We concluded that this could be its own type considering that for every play mode, a different behavior is to occur. Therefore, instead of giving the responsibility to that method, we should use polymorphism in order to do this.

2. We created an interface called IPlayMode to reflect the type we want to extract from the method discussed above.

3. We added a field IPlayMode playModeObject to song.cpp as a private attribute.

4. We identified the problematic points that could be extracted from the method. The main issue being the switch statement of play modes (line 201), which we extracted using the *Extract*

*Method[6]* refactoring; creating the new method processPlayMode(..) which is called inside processNextBuffer() instead of the initial switch statement.

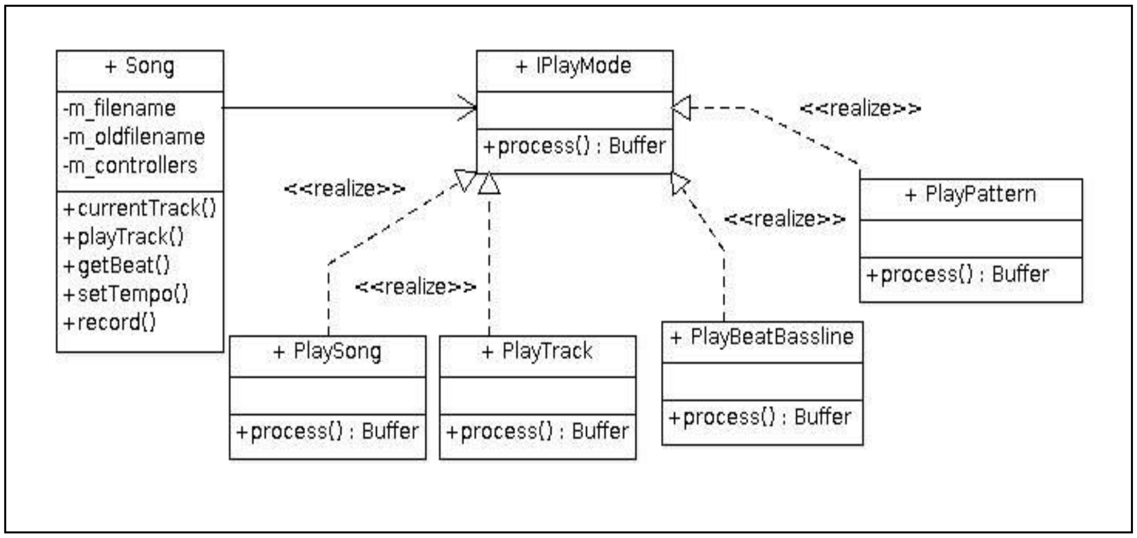
5. This lead us to extend PlayMode with 4 sub-classes (PlaySong, PlayTrack, PlayBeatBassLine, PlayPattern).

6. We then added the method process(..) inside the parent class PlayMode, which is an abstract method overridden by the child method using the *Move Method[7]* refactoring. Now the behavior that goes inside these overridden methods, were fetched from the processPlayMode(..) we have extracted in Step3. To do this, we used the *Replace Conditional with Polymorphism[8]* refactoring. For each branch of the switch statement (line 201), we removed the code and placed it in the appropriate method process(..) in the corresponding sub-class. We then replace the removed code in the switch statement, by calls to the corresponding child object (note that we are not done here as this will be further refactored in the steps listed below).

7. Now, it is still not very convenient to have calls to different sub-class objects in the initial switch statement. Therefore, inside the old processPlayMode(..), we have created an abstract query to the method process(..) to an object of static type IPlayMode, which will be resolved during runtime and return the output for the desired playmode (because of the polymorphic call).

8. Finally, the play mode was passed as an argument in the constructor of song. Therefore, we use an array of IPlayMode objects in which each of the slots corresponds to the dynamic type of the sub-classes. When an object of type song is created, the field IPlayMode will see its dynamic type assigned to the corresponding mode passed as an argument (this could be further refactored in the future). Therefore, upon calling processNextBuffer(), the call to playModeObject.process() will be dispatched to the run time type of playModeObject.

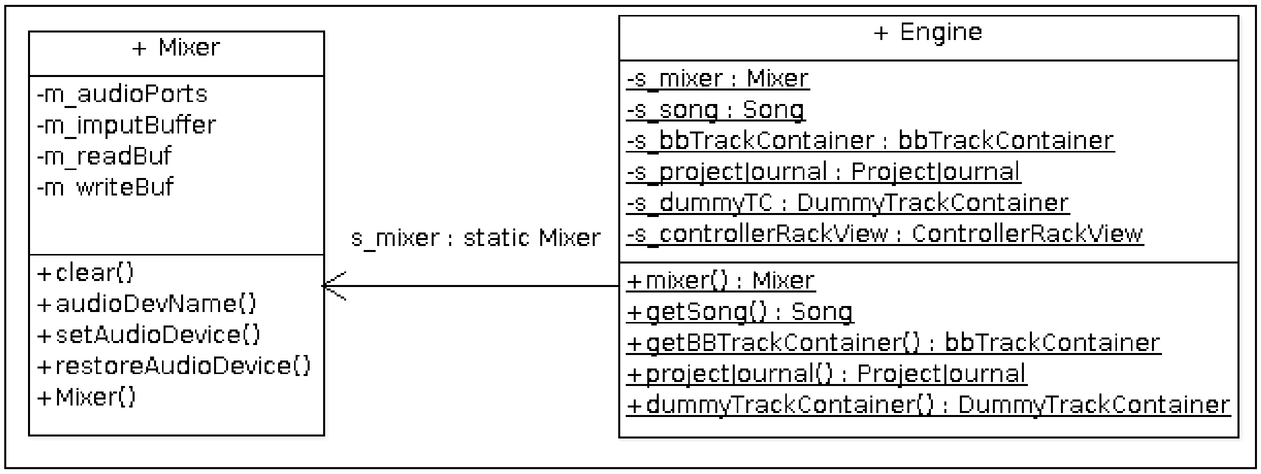
**Refactored state of song.cpp**



To conclude, this refactoring reduced the length of class song.cpp and removed the Long Method processNextBuffer() which now simply delegates the responsibility to an object of static type IPlayMode and the call to the actual process(..) is done to the appropriate dynamic type of playModeObject. Using polymorphism removed the if-else statements which make us think that, using polymorphism appropriately will in most circumstances solve bad smells associated to long methods and switch statements.

**[2] Long Method (Constructor)**

As we saw, in the above code smell, cohesion is greatly reduced when we are facing code smells such as long method or long class. Class Mixer, in the LMMS system, presents us a special case of Long Method: a very lengthy constructor (Mixer.cpp, line 294-381).


+ Mixer
-m_audioPorts
-m_imputBuffer
-m_readBuf
-m writeBuf
+clear()
+audioDevName()
+setAudioDevice()
+restoreAudioDevice()
+Mixer()
s_mixer : static Mixer
+ Engine
-s_mixer : Mixer
-s_song : Song
-s_bbTrackContainer : bbTrackContainer
-s_projectJournal : ProjectJournal
-s_dummyTC : DummyTrackContainer
-s_controllerRackView : ControllerRackView
+mixer() : Mixer
+getSong() : Song
+getBBTrackContainer() : bbTrackContainer
+projectJournal() : ProjectJournal
+dummyTrackContainer() : DummyTrackContainer

We investigated the constructor to find out why it is so long. The reason is that, there is a bunch of nested if-else statements that will generate a Mixer with particular configurations. It appears that they decided to use this approach in order to solve the issue of telescoping constructor anti-pattern [9] (i.e instead of having multiple constructor based on parameters input, they access a configuration and they use a single constructor that generates a mixer based on the set of parameters found in the configuration). Because of this, we found that having a MixerBuilder (using the Builder Pattern [10]) would make much more sense as this would give the responsibility to the Builder to generate the appropriate Mixer and not the responsibility of the Mixer itself. Therefore this will increase cohesion in the Mixer class and eliminate the Long Method (constructor) smell.

**Steps to fix the Large Method (Constructor) Code Smell**

1. We extracted the code present in the if-else statements and the for-loops into separate methods to which we pass the parameters that were being used as arguments. These methods reflect the parts that are being "constructed" in the original constructor.

2. Still the constructor is not cohesive at all because it deals with bunch of calls to the newly created method. Therefore, we decided to create a class, of type/named MixerBuilder that has the responsibility to build these parts.

3. To incrementally refactor this constructor using the MixerBuilder pattern, we moved the methods that were responsible for the creation of mixer parts to the MixerBuilder and give them the respective name: buildMixer(), buildChannels(), buildFifo() and buildWorkers() (they have a 1-1 mapping to the parts that were being built in the initial constructor).

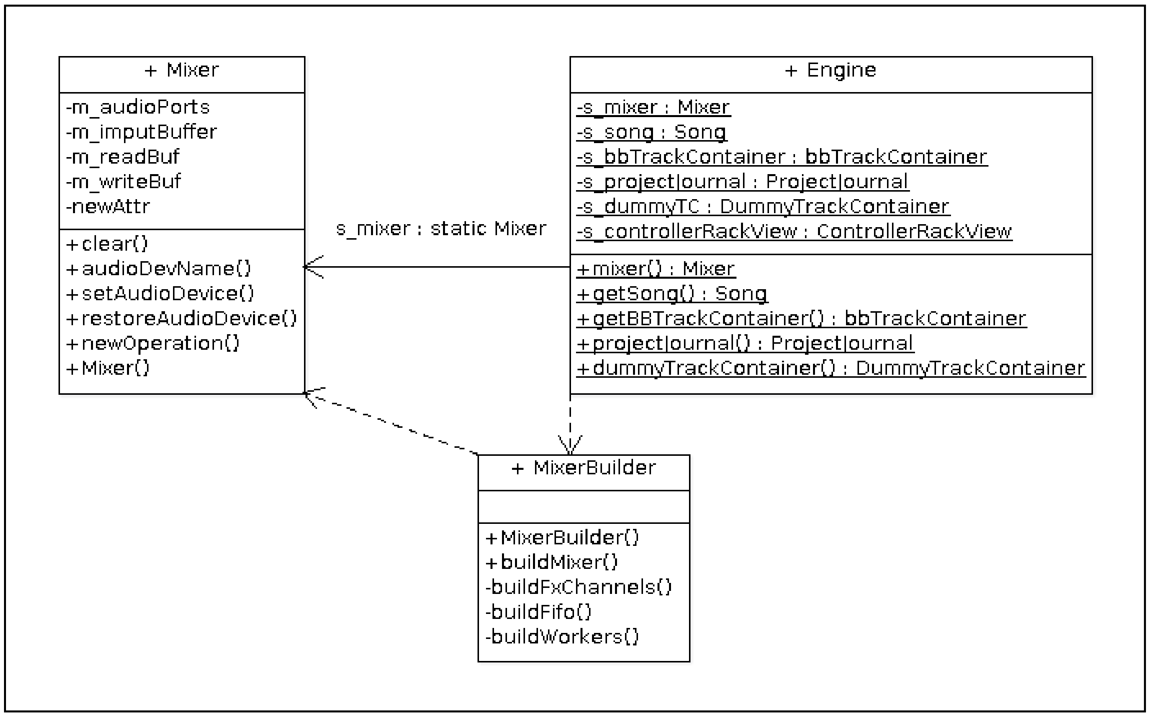
4. The next step was to add mutators and accessors to the Mixer class in order to change the fields that correspond to the parts (m_workers, m_fifo, _fx_channel_jobs).

5. Then we replaced the direct field access and modifications inside the methods of MixerBuilder to be able to perform the modification of a Mixer object.

6. At this point, we eliminated the calls to the methods that are in the initial constructor of Mixer as there is no more need for such calls (since the assembly of Mixer object is no more handled by the MixerBuilder) which increased cohesion of class Mixer and reduced the length of the constructor.

7. Finally, we coupled Engine (who originally handled the call to the constructor of a Mixer object) to the MixerBuilder.

<u>**Refactored state of Mixer.cpp**</u>



To conclude, we managed to refactor the long constructor by applying the Builder pattern which removes the responsibility of the constructor to instantiate specific Mixers. In the future, we could further refactor this class by having the MixerBuilder as a singleton considering the Engine needs only 1 MixerBuilder and there are no Mixer to be invoked concurrently.

## 3. Specific Refactorings to be implemented in M4

For milestone four, we are planning to deliver refactoring discussed in section 2 (ref: Song.cpp & Mixer.cpp Long Class & Long Method code smells). Instead-of copy-pasting the entire class, we prefer focusing on this method considering the class itself as a “Long Class “code smell.

Code to be modified:

Song.cpp [processNextBuffer()] ~line 191

```
void song::processNextBuffer()
{
	if( m_playing == false )
	{
		return;
	}

	TrackList track_list;
	int tco_num = -1;

	switch( m_playMode )
	{
		case Mode_PlaySong:
			...
			break;

		case Mode_PlayTrack:
			...
			break;

		case Mode_PlayBB:
			...
			break;

		case Mode_PlayPattern:
			...
			break;

		default:
			return;

	}

	if( track_list.empty() == true )
	{
		return;
	}

	...

	if( check_loop )
	{
        if( m_playPos[m_playMode] < tl->loopBegin() ||
                        m_playPos[m_playMode] >= tl->loopEnd() )
        {
        ...
        }
    }


    while( total_frames_played < engine::mixer()->framesPerPeriod() )
    {
        if( current_frame >= frames_per_tick )
        {
            if( ticks >= MidiTime::ticksPerTact() )
            {
                if( m_playMode == Mode_PlayBB )
                {
...
                }
                else if( m_playMode == Mode_PlayPattern &&
                        m_loopPattern == true &&
                        tl != NULL &&
                        tl->loopPointsEnabled() == false )
                {
                }

                // end of played object reached?
                if( m_playPos[m_playMode].getTact() + 1 >= max_tact )
                {
                }
            }
            m_playPos[m_playMode].setTicks( ticks );

            if( check_loop )
            {

                if( m_playPos[m_playMode] >= tl->loopEnd() )
                {

                }
            }
            else
            {

            }
        }


        if( last_frames == 0 )
        {

        }

        if( last_frames < played_frames )
        {
        }

        if( (f_cnt_t) current_frame == 0 )
        {
            if( m_playMode == Mode_PlaySong )
            {
            }

            for( int i = 0; i < track_list.size(); ++i )
            {

            }
        }

    }
}
```

## 4. References

| | |
|---|---|
| [1] | UI Gripes: http://sourceforge.net/p/lmms/mailman/message/31913561/ |
| [2] | UI Rework: http://sourceforge.net/p/lmms/mailman/message/31853352/ |
| [3] | New UI: http://sourceforge.net/p/lmms/mailman/message/31832076/ |
| [4] | Song Editor Look: http://sourceforge.net/p/lmms/mailman/message/32009515/ |
| [5] | http://sourcemaking.com/refactoring/replace-type-code-with-state-strategy |
| [6] | http://sourcemaking.com/refactoring/extract-method |
| [7] | http://sourcemaking.com/refactoring/move-method |
| [8] | http://sourcemaking.com/refactoring/replace-conditional-with-polymorphism |
| [9] | https://kodelog.wordpress.com/tag/telescopic-constructor-pattern/ |
| [10] | Design Patterns: Elements of Reusable Object-Oriented Software; Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, p97 (C3) |
| [11] | Design Patterns: Elements of Reusable Object-Oriented Software; Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, p315 (C5) |

# Appendix A

Additional Code Smells & Refactorings needed:

| ID | Location | Type | Action to take |
|---|---|---|---|
| 1 | Song.cpp | Replace type code with subclasses | Switch statement with enumerator that depending on value has different behavior. This can be refactored using a strategy pattern [11]. |
| 2 | Mixer.cpp, line 551, renderNextBuffer() | Long Method | Break down to smaller more cohesive methods. Use a builder to construct all the smaller, necessary parts. |
| 3 | engine.cpp/engine.h | Static Class (not actual formal code smell, but problematic) | This is a problem: a static class would imply that we have global visibility to its attributes (true in this case), and because nothing like a singleton is used, we are not given an interchangeable interface. If we were to change this, then turning the class to a singleton would probably be better. Turning the engine to an object, and using dependency injection would be the best solution, but would have tremendous rippling changes.<br><br>If this is refactored properly, it would mean that different projects can be open at the same time if required by the user, adding more power to the application. |
| 4 | track.h / track.cpp | Different class definitions in same file | Separate the class definitions. Since other files reference these classes in the same file, put at the top of the header, includes for their definitions. |
| 5 | DataFile.cpp, upgrade(), line 295 | Long Method | This is an upgrading mechanism. LMMS has been around for a while, and its domain data has been persisted in different ways. Because newer versions may store information in a 'new' way, we have to provide a mechanism to understand the older ones too. This is what this function does, but can be broken down into smaller cohesive parts. |

| | | | |
|---|---|---|---|
| 6 | Song.cpp / Song.h | Inheritance abuse | Song directly inherits TrackContainer, which explodes the interface of Song. Song also is not necessarily a TrackContainer as well. This should, and could be refactored using delegation in order to reduce interface bloat, and a more cohesive Song class. |
| 7 | RemotePlugin.cpp, process(sampleFrame, sampleFrame), line 163 | Long Method | Some parts may be factored out into smaller functions. There is a lot of nested loops, and nested logic, which makes the code hard to understand and less maintainable. |
| 8 | RemotePlugin.cpp, processMessage(message), line 342 | Replace Type Code with Subclass | Different behavior exists in this method that depends on an enumeration. This can be broken down to different classes (in this case have a message class per enumeration and insert the code behavior in there) |