Performance Analysis Code Overview

There are three phases of a performance, each with a corresponding general area of code:

1) **Performance Tracking:** While the student is playing along with a score, for each note the performed sound, data is captured and, if possible, matched to the expected note data. This involves two concurrent subsystems the MusiKyoshi App is using: 1) SeeScore, which is playing back the score and informing the app when each note begins and ends, and 2) AudioKit, which provides data on the details of an audio signal, including frequency.

2) **Post-Performance Analysis and Grading**: Once the performance is done - when either the song stops or the student manually stops playback - each performed note is graded for pitch, rhythm attack (how close to "on the beat" did the student play), and note duration accuracy. For each note, each of these categories gets a grade. If there are problems, the each category is also given a numeric "severity" score (e.g., very flat gets a worse score that slightly flat). The code also detects if a note is a partial of the expected pitch. An overall score is also assigned.

3) **Issue Sorting**: Sorting through all the issues, deciding which is the worst, and offering help. It is possible to select any of the three performance categories - pitch, attack, or duration - as the only criteria to consider. Or, the worst of any category can be the selecting criteria. Or, the sum of all of the severity scores. When there is a video for the detected issue, the Note is highlighted on the score, and the video is presented and playback begins.

The intent is make it easy to "tune" the software by changing several values:
- The frequency thresholds for what is considered correct pitch, slightly flat or sharp, etc.
- The timing thresholds for what is considered correct timing, slightly early or late, etc.
- The severity scores for each degree of variance from the expected performance.
- The criteria for choosing what is the worst type of issue (could do multiple passes, etc.).

The main components are:

PerformanceTrackingMgr: manages both the expected note info (PerformanceNotes) and the actual sounds produced by the student (PerformanceSounds). Links these together if they are related, so individual notes can be graded.

PerformanceAnalysisMgr: responsible for building the tables used to identify note pitches, the accuracy zones for grading pitches, the tables used for identifying Partials, and more. Also responsible for post-performance grading, where Pitch and Rhythm Analyzers grade each note for pitch and rhythmic accuracy.

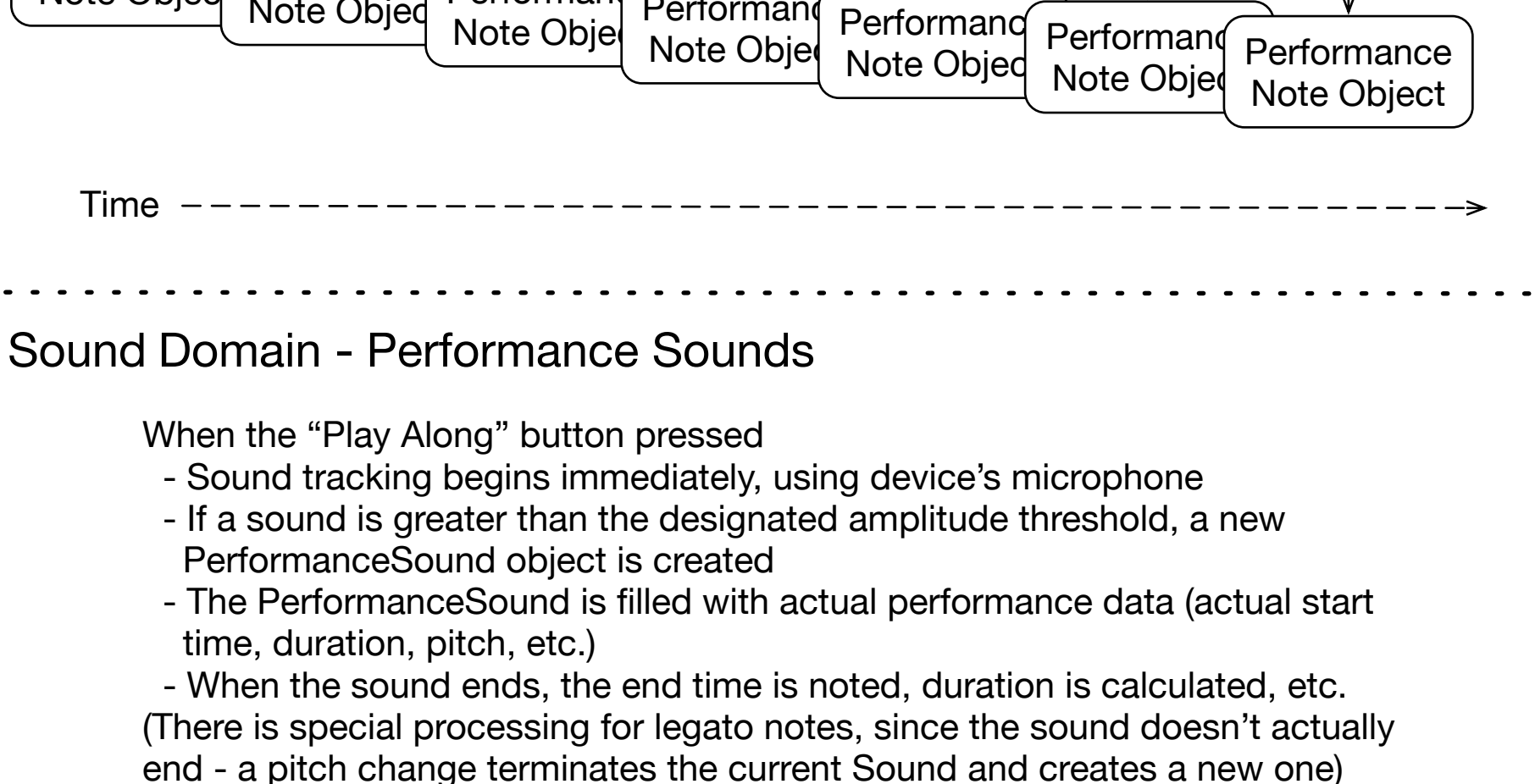
PerformanceIssueMgr: visits each (graded) PerformanceNote and creates a separate PerformanceIssue object for it. These issues can be prioritized and sorted by different criteria, such as pitch, rhythm, highest individual, etc. When this phase is finished, the array of PerformanceIssues is sorted from the most to least severe issue.

VideoHelpView and related code: Using info stored in the worst PerformanceIssue, can direct the SeaScore view to move to the location of the offending note, highlight the note if desired, and launch the video assigned to the issue. (The view itself is perhaps just temporary - proof of concept. But the other infrastructure code is not intended to be.)

Score Domain - Performance Notes

When the "Play Along" button pressed, after intro, SeaScore plays back Score
- As each note begins, SeaScore calls the "start" event handler in the VC
- A new PerformanceNote object is created
- The PerformanceNote is filled with:
  - MusicXML note data (expected start time, duration, pitch, etc.)
  - The X and Y offsets, on the SeaScore ScrollView, of the drawn note
  - (and more . . .)
- When the note ends, SeaScore calls the "end" event handler in the VC

The result: A PerformanceNote object with expected performance data for each note in the score, plus data needed to draw on the SeaScore view (if needed)

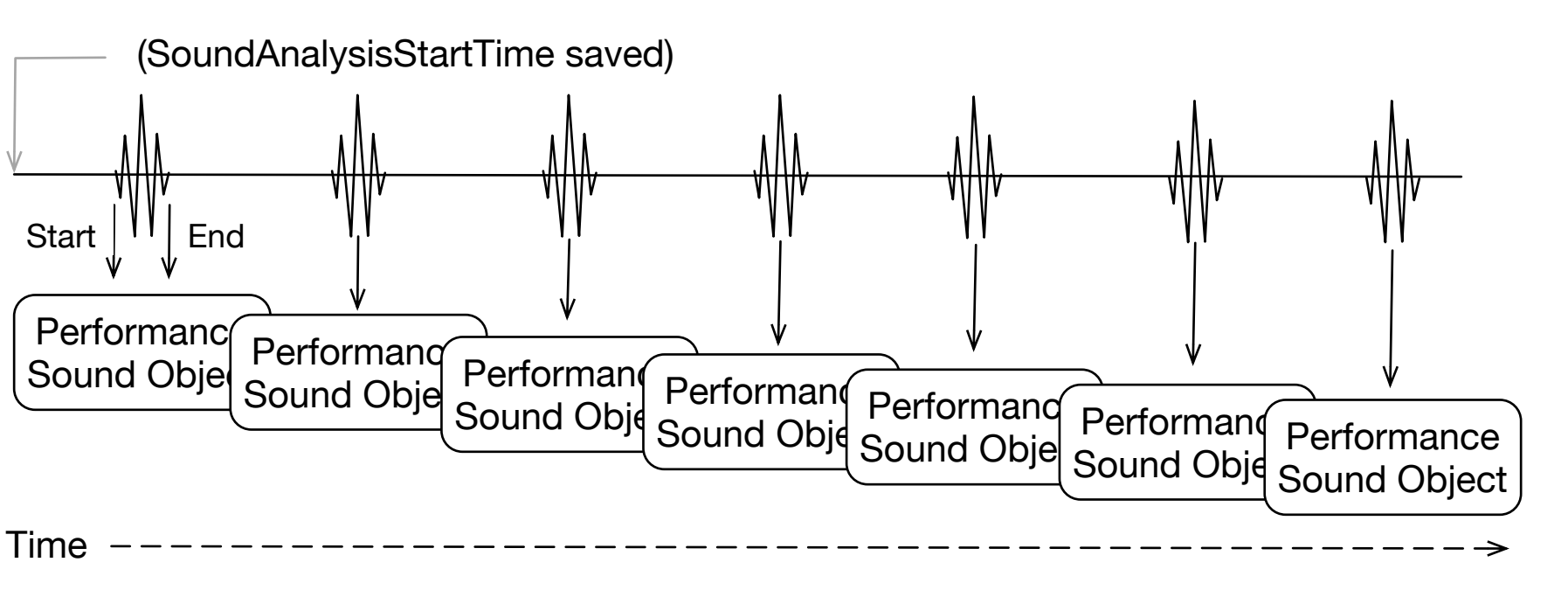


Sound Domain - Performance Sounds

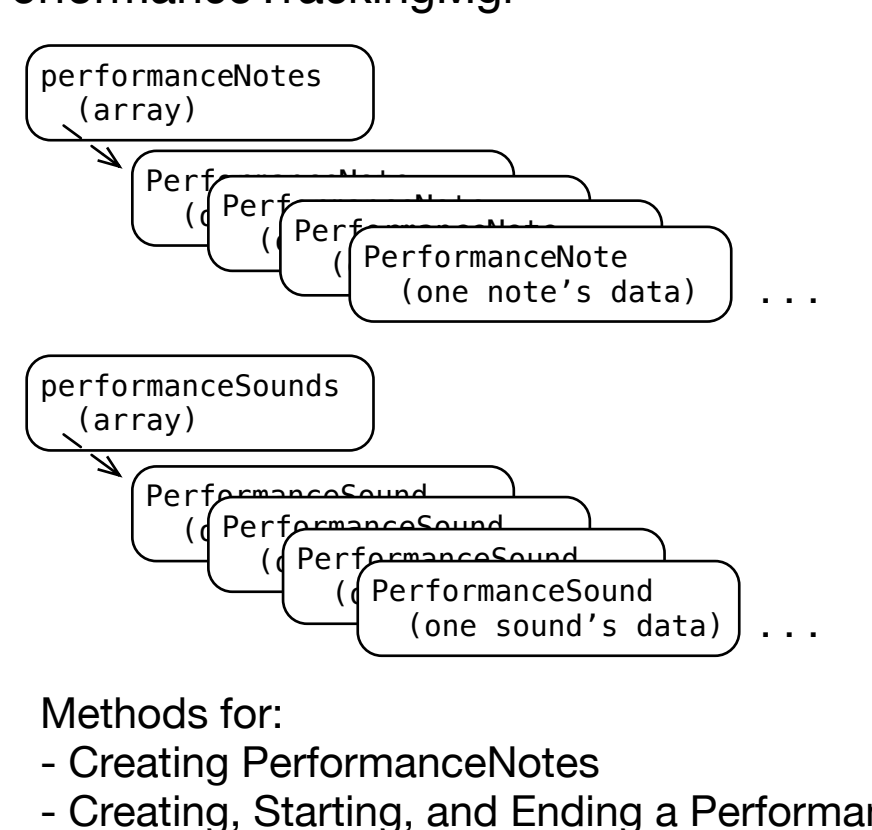
When the "Play Along" button pressed
- Sound tracking begins immediately, using device's microphone
- If a sound is greater than the designated amplitude threshold, a new PerformanceSound object is created
- The PerformanceSound is filled with actual performance data (actual start time, duration, pitch, etc.)
- When the sound ends, the end time is noted, duration is calculated, etc.

(There is special processing for legato notes, since the sound doesn't actually end - a pitch change terminates the current Sound and creates a new one)

The result: A PerformanceSound object with actual performance data for each sound produced by the student



PerformanceTrackingMgr



Exposes *performanceNotes* array for access by PerformanceAnalysisMgr and PerformanceIssuesMgr

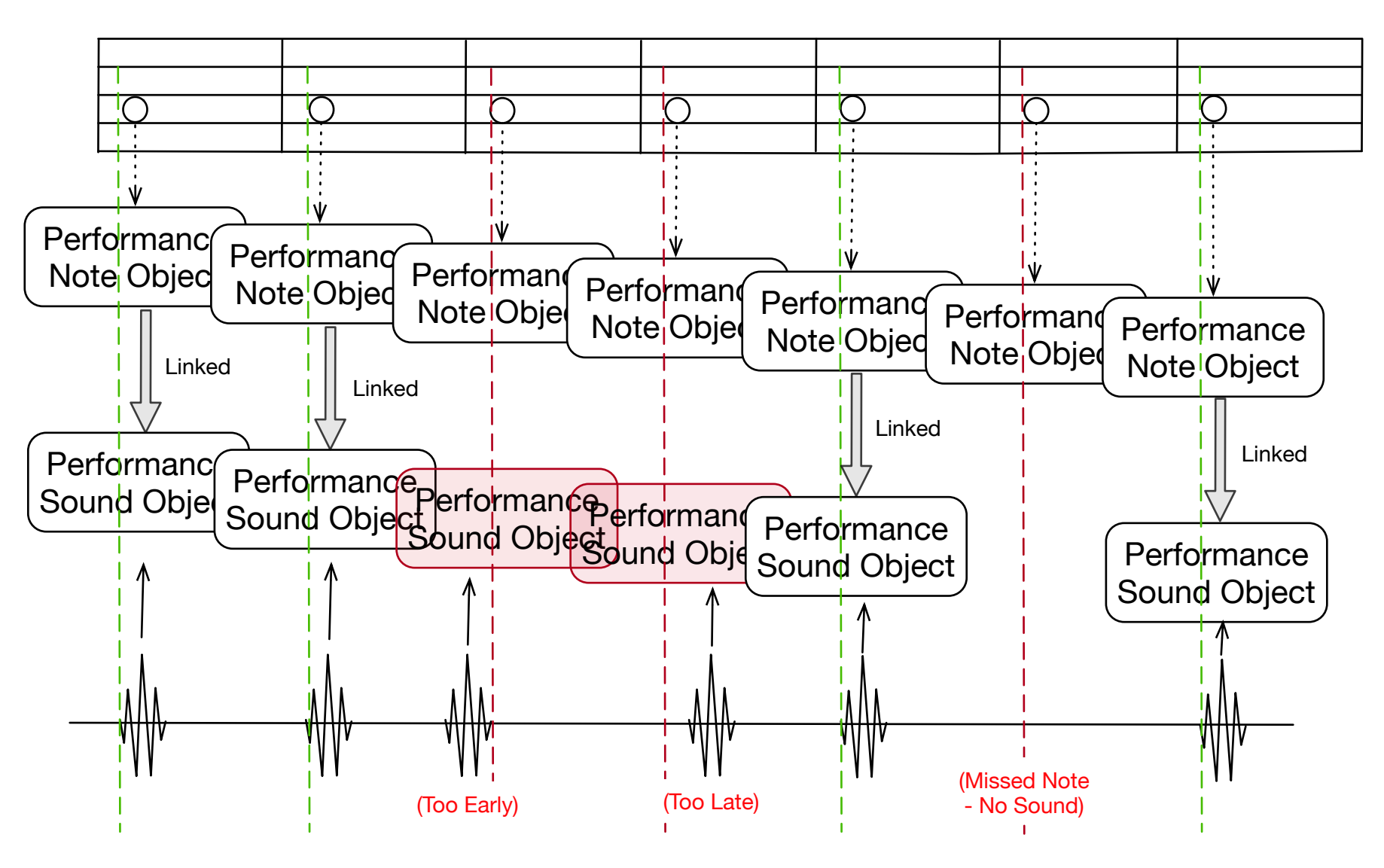
Methods for:
- Creating PerformanceNotes
- Creating, Starting, and Ending a PerformanceSound

- Retrieving a PerformanceNote by various criteria
- Retrieving a PerformanceSound by various criteria
- Linking a PerformanceNote to a PerformanceSound

- Setting the SongStart
- Setting the SoundAnalysisStart

Linking Performance Sounds to Performance Notes

When there is an unlinked PerformanceNote, and a sound occurs within the same time window as the PerformanceNote's expected start time, the note and sound are linked. This allows comparison of expected and actual values for attack, pitch, duration, etc. (Sounds outside the timing threshold are not linked - threshold "forgiveness" can be adjusted.)



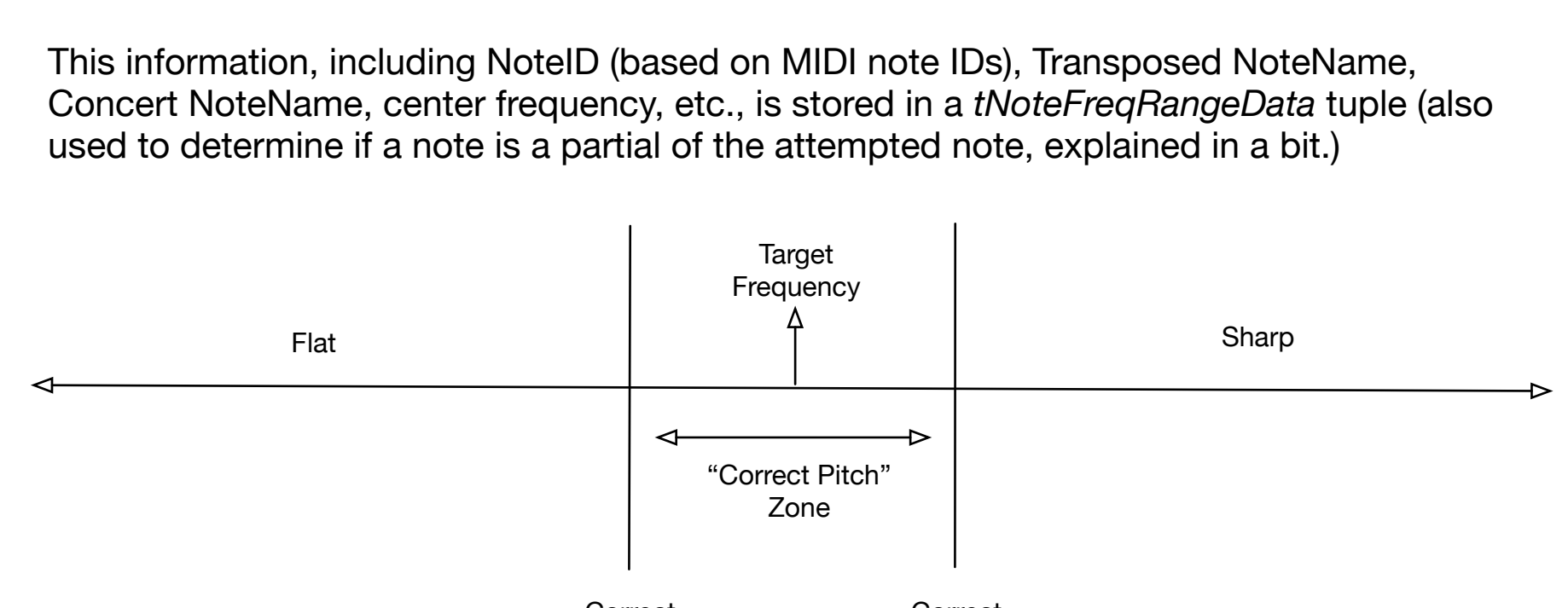
Post-Performance Analysis

For comparing a Note's performed pitch to the expected frequency, the primary acceptable frequency range is determined by applying the correctPitchPC percentage (a member of *pitchAndRhythmTolerances*) to the target frequency:

```
TargetFrequency x correctPitchPC   == Correct Frequency Lower Boundary
TargetFrequency / correctPitchPC   == Correct Frequency Upper Boundary
```

This is usually a difference of around 3%. (The inverse, or 97%, is used in the equation above).

This information, including NoteID (based on MIDI note IDs), Transposed NoteName, Concert NoteName, center frequency, etc., is stored in a *tNoteFreqRangeData* tuple (also used to determine if a note is a partial of the attempted note, explained in a bit.)

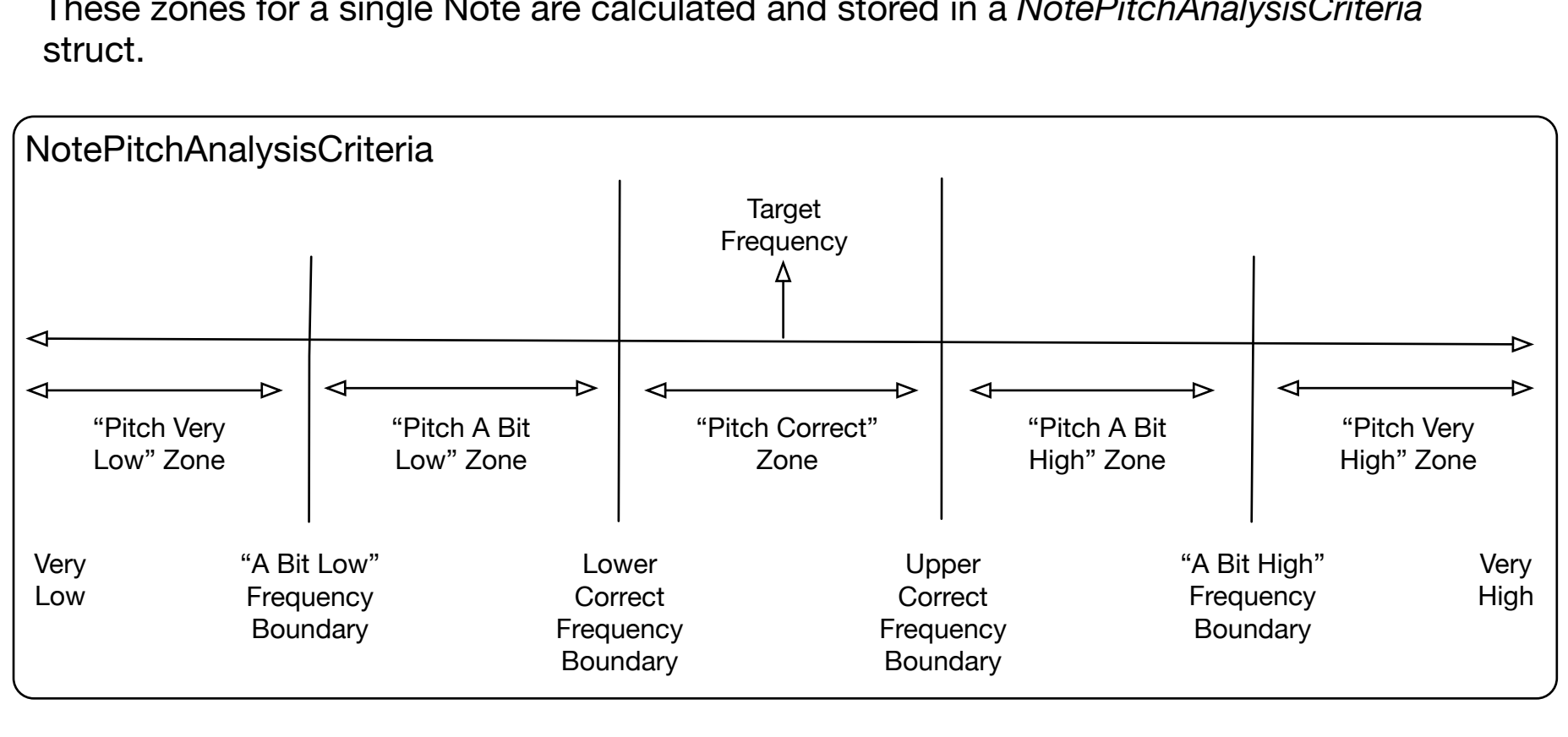


For actually grading a Note's performed pitch, there are five zones: CorrectPitch (discussed earlier), Slightly Low, Slightly High, Very Low, and Very High, shown below.

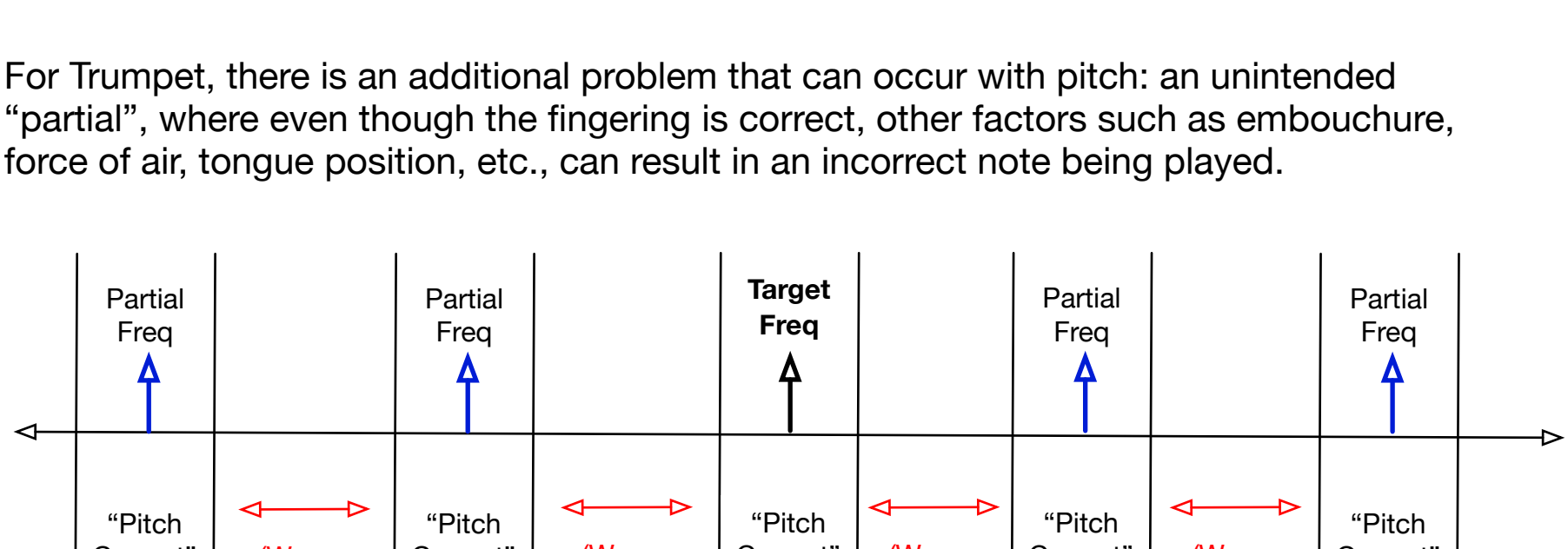
The boundaries for these Zones are calculated by applying percentages to the target frequency of the Note. These percentages are also members of *pitchAndRhythmTolerances*.

```
TargetFrequency x aBitToVeryPercentage   == "A Bit Low"  Frequency Boundary
TargetFrequency / aBitToVeryPercentage   == "A Bit High" Frequency Boundary
TargetFrequency x veryBoundaryPercentage == "Very Low"   Frequency Boundary
TargetFrequency / veryBoundaryPercentage == "Very High"  Frequency Boundary
```

These zones for a single Note are calculated and stored in a *NotePitchAnalysisCriteria* struct.



For Trumpet, there is an additional problem that can occur with pitch: an unintended "partial", where even though the fingering is correct, other factors such as embouchure, force of air, tongue position, etc., can result in an incorrect note being played.



For each fingering (Position), the *tNoteFreqRangeData* for each partial is stored in an array in a *TrumpetPartialsForPosition* class.

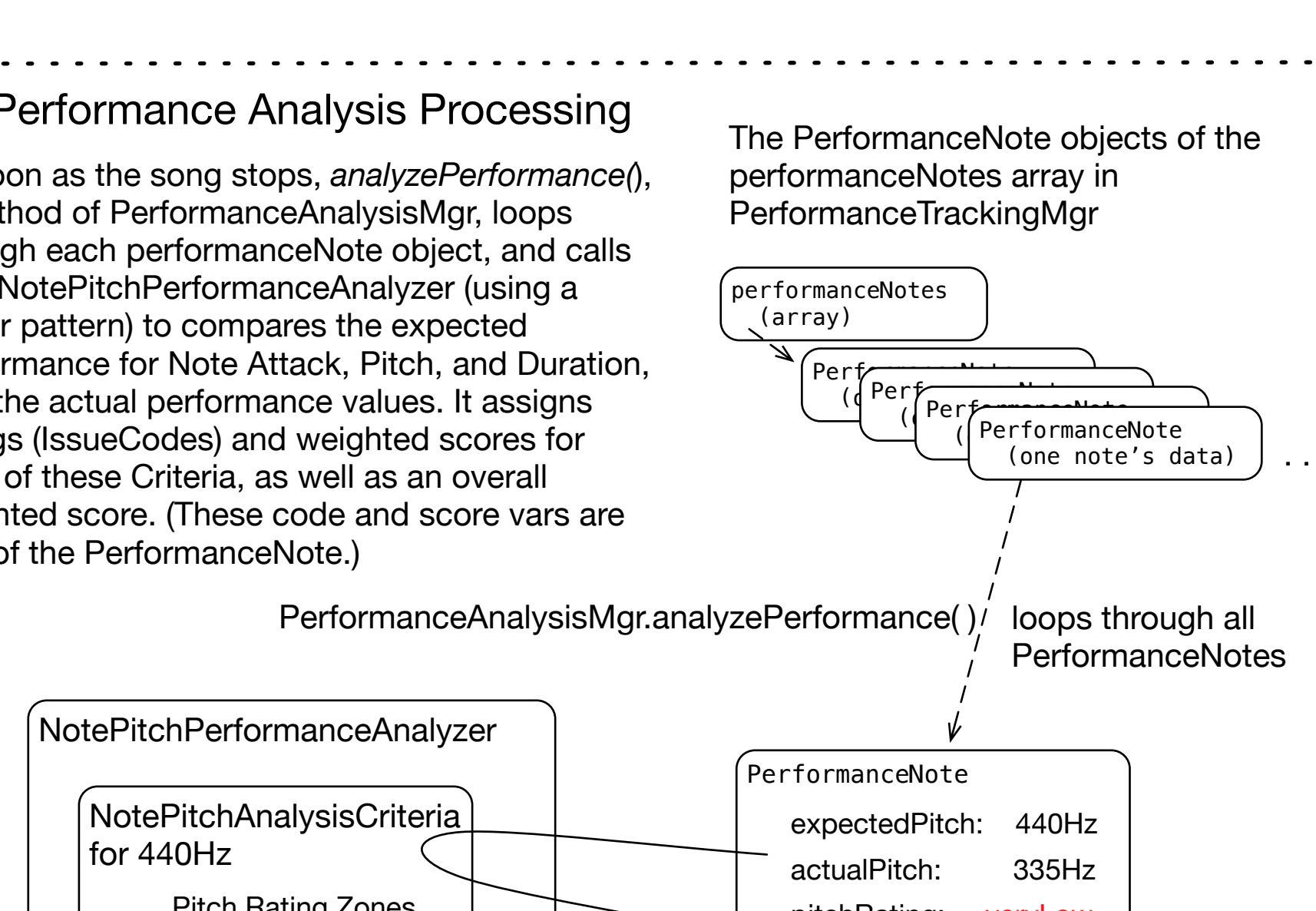
There are seven fingerings (positions) on the trumpet, so seven *TrumpetPartialsForPositions* are created and managed in the *TrumpetNotePartialsTable*.

The method *isThisFreqAPartialOfThisNote*() can be called to determine if a wrong pitch is actually a partial of the correct pitch. This issue requires different help than a low or high note due to other reasons.

Post-Performance Analysis Processing

As soon as the song stops, *analyzePerformance*(), a method of PerformanceAnalysisMgr, loops through each performanceNote object, and calls on a NotePitchPerformanceAnalyzer (using a visitor pattern) to compares the expected performance for Note Attack, Pitch, and Duration, with the actual performance values. It assigns ratings (IssueCodes) and weighted scores for each of these Criteria, as well as an overall weighted score. (These code and score vars are part of the PerformanceNote.)

The PerformanceNote objects of the performanceNotes array in PerformanceTrackingMgr



(Note: a low score is best. E.g., 0 is perfect, 2 is okay, 6 is pretty bad, etc.)

PerformanceIssues

A PerformanceIssue object is created to represent the worst issue of one PerformanceNote. What is considered the worst issue is decided by the *current Sort Criteria* (explained on the next page); so even though a given Note might have issues with pitch *and* timing, only one issueScore and issueRating field is stored as *the* PerformanceIssue.

For example, for the PerformanceNote below, if the sortCriteria is .byPitch, the other fields are ignored and only the rating and score for pitch are used.

(Note that the PerformanceNote object itself is not changed in this process.)

It is also possible at this stage, once the issueRating is established, to look up the videoID for this particular issue and, if there is one, assign it to the videoID field.

```
PerformanceNote   ID: 1

 attackRating:   .correct
 attackScore:    0
 durationRating: .correct
 durationScore:  0
 pitchRating:    .verySharp
 pitchScore:     6
 overallScore:   8

(All other data in the
PerformanceNote is also
available, if needed. E.g.,
target and actual pitch info,
user-friendly Note Name, etc.)
```

```
PerformanceIssue

issueScore:          6
issueRating:         .verySharp
performanceNote ID:  1
videoID:             kVidID_aBitSharp
issueType:           .pitch
```

Call to map IssueRating to VideoID. (Can return noneAvailable.)

Sorting Results by Issue Type and Severity

```
PerformanceNote   ID: 1

 attackRating:   .correct
 attackScore:    0
 durationRating: .correct
 durationScore:  0
 pitchRating:    .verySharp
 pitchScore:     6
 overallScore:   8
```

```
PerformanceNote   ID: 2

 attackRating:   .veryLate
 attackScore:    6
 durationRating: .aBitShort
 durationScore:  2
 pitchRating:    .aBitSharp
 pitchScore:     2
 overallScore:   10
```

```
PerformanceNote   ID: 3

 attackRating:   .aBitLate
 attackScore:    2
 durationRating: .tooShort
 durationScore:  4
 pitchRating:    .correct
 pitchScore:     0
 overallScore:   6
```

To create the *PerformanceIssues,* in the last stage of *analyzePerformance*(), the *scanPerfNotesForIssues(sortCriteria)* method of *PerformanceIssueMgr* is called. This produces a sorted array of *PerformanceIssues*. The *sortCriteria* param determines which score is used (for all notes) to determine the worst issues. The options are pitch, attack, duration, highest of each of these, or the sum of all.

For the three notes shown here, using different sorting criteria would create the following results in the resulting *sortedPerfIssues* array, worst to last (again, *lowest* score is best):

Sorted using .byAttackRating, PerfNote IDs: 2, 3, 1

Sorted using .byDurationRating, PerfNote IDs: 3, 1, 2

Sorted using .byPitchRating, PerfNote IDs: 1, 2, 3

Sorted using .byIndividualRating, PerfNote IDs: 1, 2, 3

Sorted using .byOverallRating, PerfNote IDs: 2, 1, 3

Choosing which sort criteria to use at what times - or coming up with additional methods to rate/score/sort - is something we can do as we try all of this out and tune it.

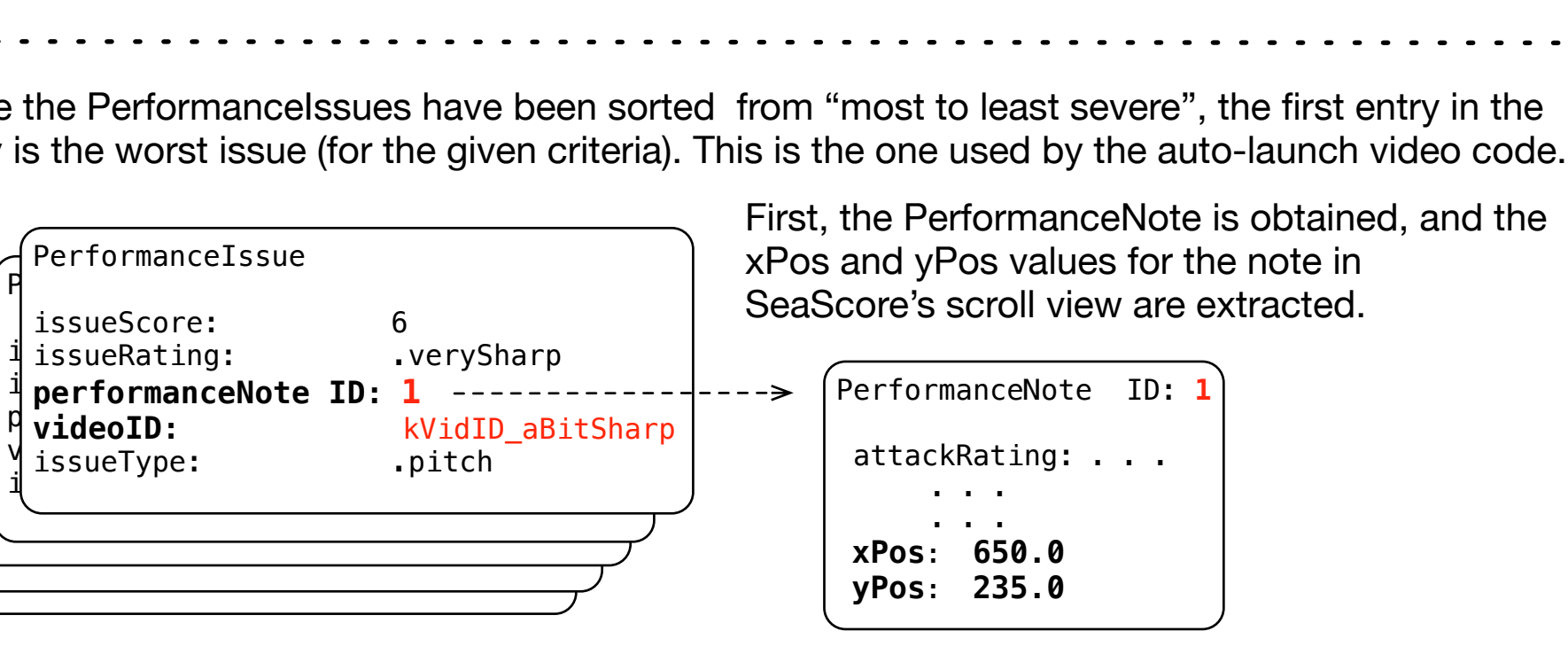
Since the PerformanceIssues have been sorted from "most to least severe", the first entry in the array is the worst issue (for the given criteria). This is the one used by the auto-launch video code.

```
PerformanceIssue

issueScore:          6
issueRating:         .verySharp
performanceNote ID:  1
videoID:             kVidID_aBitSharp
issueType:           .pitch
```

First, the PerformanceNote is obtained, and the xPos and yPos values for the note in SeaScore's scroll view are extracted.

```
PerformanceNote   ID: 1

 attackRating: . . .
     . . .
     . . .
 xPos:  650.0
 yPos:  235.0
```

SeaScore's scroll view is told to scroll to the PerformanceNote's xPos, with an adjustment to place the note near, but not on, the edge of the view.



Then the OverlayView is told to move the red highlight circle layer to xPos, yPos.

Next, the videoID is obtained, and - if not .noneAvailable - sent to the VideoHelpView.

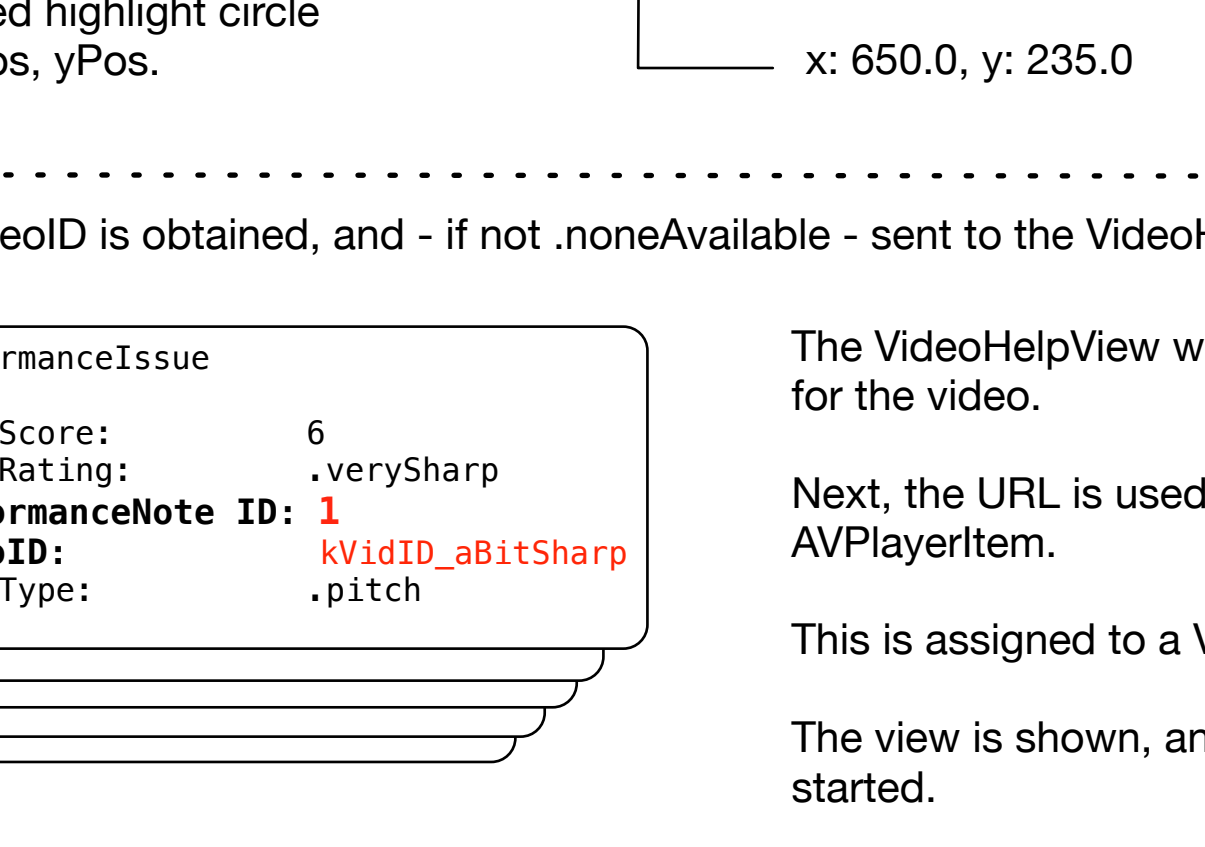
```
PerformanceIssue

issueScore:          6
issueRating:         .verySharp
performanceNote ID:  1
videoID:             kVidID_aBitSharp
issueType:           .pitch
```

The VideoHelpView will obtain the URL for the video.

Next, the URL is used to create an AVPlayerItem.

This is assigned to a VideoPlayer, etc.

The view is shown, and the video is started.



Once again, I'd like to stress that a lot of this - such as setting the severity of different issues relative to each other or setting how to decide what is the worst issue - is highly and easily tweakable. I've tried to make it so it will be very easy to try something, change a few constants, recompile and try again within seconds - until the app is behaving as desired.

And, I'd like to point out once again that is what the code does now - it doesn't have to stay this way. We can change it, and we can use parts of it to produce other behaviors.

I've tried to make this a modular system, with components as decoupled as possible. Most classes/structs are unaware of object types above them and one degree removed from them.

There are exceptions. For example, the SeeScore code is responsible for playing back the MusicXML score, and generates callbacks into the enclosing ViewController (e.g., TuneExercideViewController) for startOfBar, note start and end, and others. My code requires these callbacks.

It's possible to extract some of the code in TuneExerciseViewController, including both my code and some existing code (which no longer needs to be in there, if we move forward with my vision of performance analysis) and move it to one my my new files, to both reduce the size and complexity of TuneExerciseViewController, and to make the code more easily adaptable to other views / view controllers.