An Architecture of Music: The Digital Scriptorium & Orchestra

Deconstructing the design of a complete music notation and playback system

The Foundation: A Language for Music

Before music can be written or played, its language must be defined. The MuBasics unit serves as the system's foundational lexicon, translating the abstract concepts of music theory into concrete data structures.

Vertical (Pitch)

```
type
  HeightType = (CLowest, …, c5, NoHeight); // 10 octaves of notes
  KindType = (DoubleFlat, Flat, Natural, Sharp, DoubleSharp);
```

Horizontal (Time)

```
type
  StartTimeType = longint;
  DurationType = longint;
  FaceValueType = (HemiQuaver, …, Maxima, NoFaceValue);
  AccentType = byte; // Constants: OnBar = 0, OnBeat = 1, etc.
```

Expressing Musical Ideas: Tonality

```
type
  TonalityType = (NoTonality, TonalEsm, TonalDes, … , TonalFis);
```

Expressing Musical Ideas: Meter

```
type
  TimeDefObj = object
    CntNumb : CntNumbType; // Nominator
    CntUnit : CntUnitType; // Denominator
  end;
```

The KeyTable: An Encyclopedia of Harmony in Code

The system embeds a complete model of musical keys, their signatures, and their relationships directly into a constant data structure. This `KeyTable` is the engine for all tonal logic.

```
const
  KeyTable : array[TonalityType] of KeyDescr = (
    ...
    { Am }  (Name : 'Am';  Signature : ' 0';  SignCount : 0;  Kind : Natural;
             KeySet : [ABeses, BCes, BisC, DEses, EFes, EisF, GAses];
             LeadingTone : GisAs;  Tonica : ABeses;  Dominant : EFes;
             SubDominant : DEses;  NextKey : TonalBesm;  PrevKey : TonalGism),
    { G }   (Name : 'G';  Signature : '1#';  SignCount : 1;  Kind : Sharp;
             KeySet : [GAses, ABeses, BCes, BisC, DEses, EFes, FisGes];
             LeadingTone : FisGes;  Tonica : GAses;  Dominant : DEses;
             SubDominant : BisC;  NextKey : TonalAs;  PrevKey : TonalFis),
    ...
  );
```

The basics of key identification.

Defines which notes belong to the key's scale.

Encodes fundamental harmonic relationships.

Establishes the Circle of Fifths, enabling intelligent transposition.

The Master Scribes: An Abstract Foundation for I/O

The architecture is built on a foundation of abstract objects. These define a common interface for core tasks like parsing, printing, and saving, allowing for different concrete implementations later. This is a classic example of "programming to an interface, not an implementation."

The Reader (MuAbsParser)

Defines the contract for reading and interpreting any musical file format.

```
ParserObj = object (ReaderObj)
  procedure ParseFile; virtual;
  procedure ReportError(ErrorTxt :
string);
  procedure ReportWarning(WarningTxt :
string);
end;
```

The Printer (MuAbsPrt)

Defines the methods for printing a score to any device, handling pagination and layout.

```
PrintObj = object
  procedure PrintPages(Plan :
PrintPlanPtr; ...); virtual;
  procedure PrintArea(XStart, YStart,
....); virtual;
end;
```

The Saver (MuAbsSav)

Provides a universal way to save the internal music representation (PiecePtr) to a file.

```
SaverObj = object (WriterObj)
  procedure SavePiece(const Piece :
PiecePtr); virtual;
end;
```

NotebookLM

Scribes with Insight: Interpreting Rhythm and Meter

The key idea: The system goes beyond simply storing notes; it analyzes their context to determine musical properties like accentuation and bar numbering automatically.

The Challenge of Accentuation

How does software understand that the first beat of a bar is strong? Or that some off-beats are stronger than others?

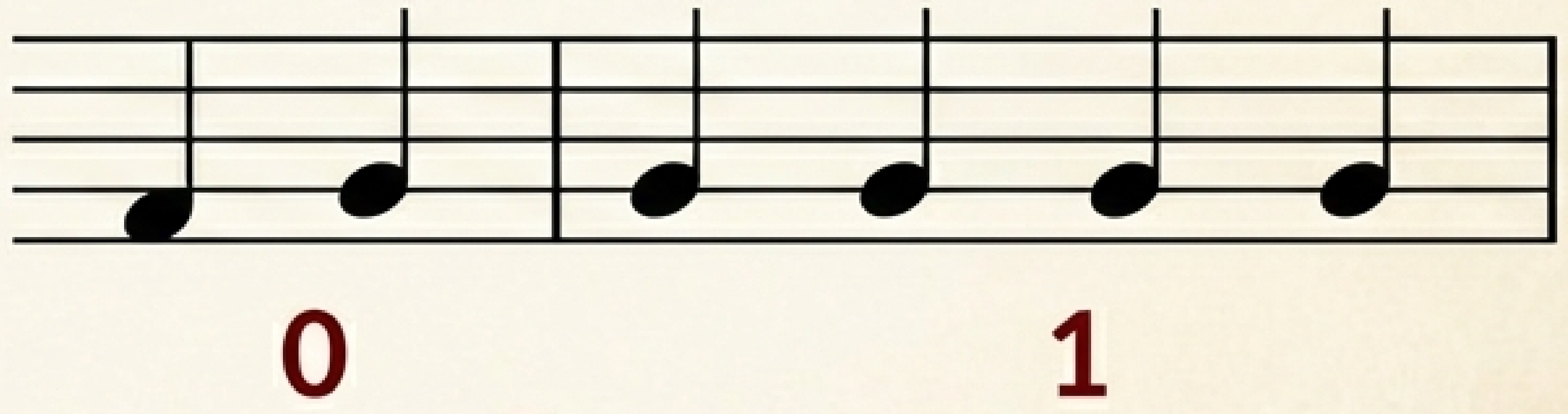
The `Accentuation` function in `MuAccent`. It mathematically determines a note's rhythmic importance based on its position within the bar, beat, and count.

```
function Accentuation(BarOffSet, BeatDuration,
      CountDuration : DurationType) : AccentType;
begin
  if BarOffSet = 0 then
    Accentuation := OnBar
  else if BarOffSet mod BeatDuration = 0 then
    Accentuation := OnBeat
  else if BarOffSet mod CountDuration = 0 then
    Accentuation := OnCount
  else ...
end;
```

The Logic of Bar Numbering

How does the system correctly number bars, especially when dealing with an upbeat (anacrusis)?

The `NumberAllBars` procedure in `MuBarNumber`. It iterates through the piece, calculates the duration of the first bar, and intelligently sets the starting bar number to 0 or 1 accordingly.



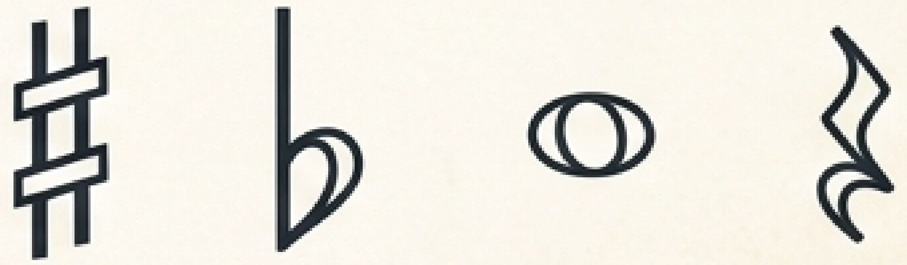
The Art of Calligraphy: Drawing Musical Glyphs

Key Idea: Every symbol on the musical score—from the elegant G-clef to the humble sharp sign—is meticulously defined as a low-level bitmap. This unit is the system's digital inkwell.

```
// from DefineGClef
DefArray : MonoArray = (
  ...
  { 19. ........|...XXXXX|X....... } $00, $1F, $80,
  { 20. ........|...XXXXX|........ } $00, $1F, $00,
  { 21. ........|..XXXXX.|........ } $00, $3E, $00,
  { 22. ........|.XXXXX..|........ } $00, $7C, $00,
  ...
);
```

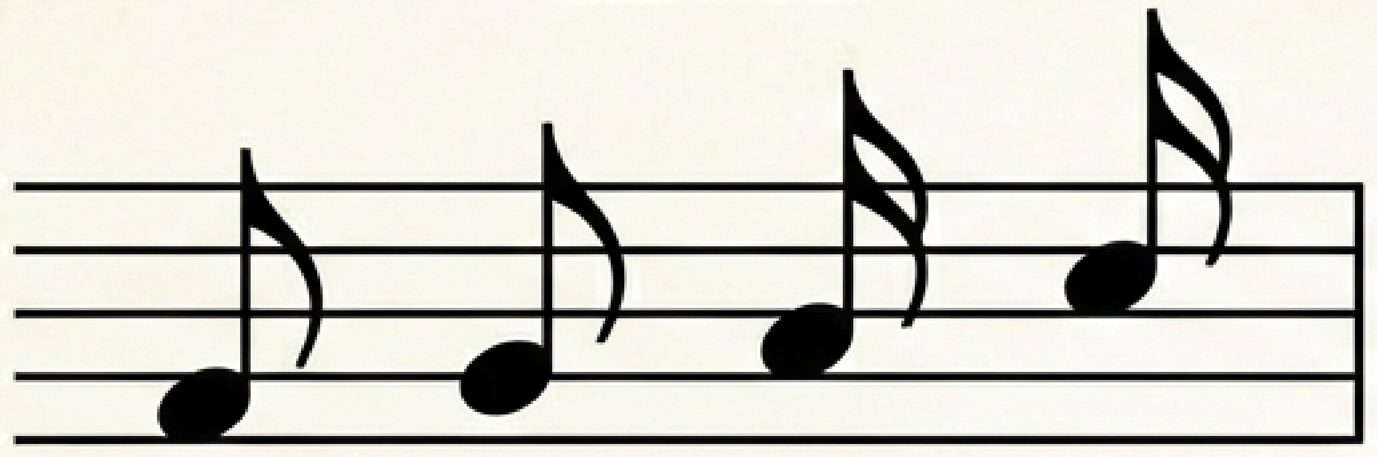
Noteheads & Rests

The system defines bitmaps for all essential symbols: Clefs (`GClef`, `FClef`), accidentals (`Sharp`, `Flat`), notes (`OpenNote`, `ClosedNote`), and rests (`CrotchetRest`, `QuaverRest`).



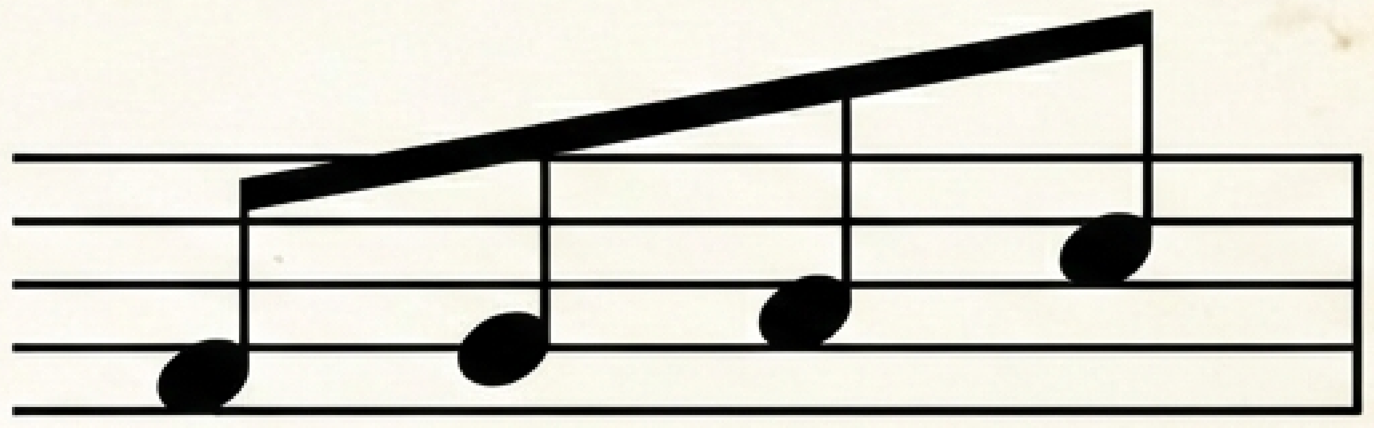
The Geometry of Beaming: A Challenge in Notation

Before



Unbeamed

After



Properly Beamed

The Challenge: Drawing rhythmically-correct beams for groups of notes is not trivial. The angle and thickness of beams depend on:

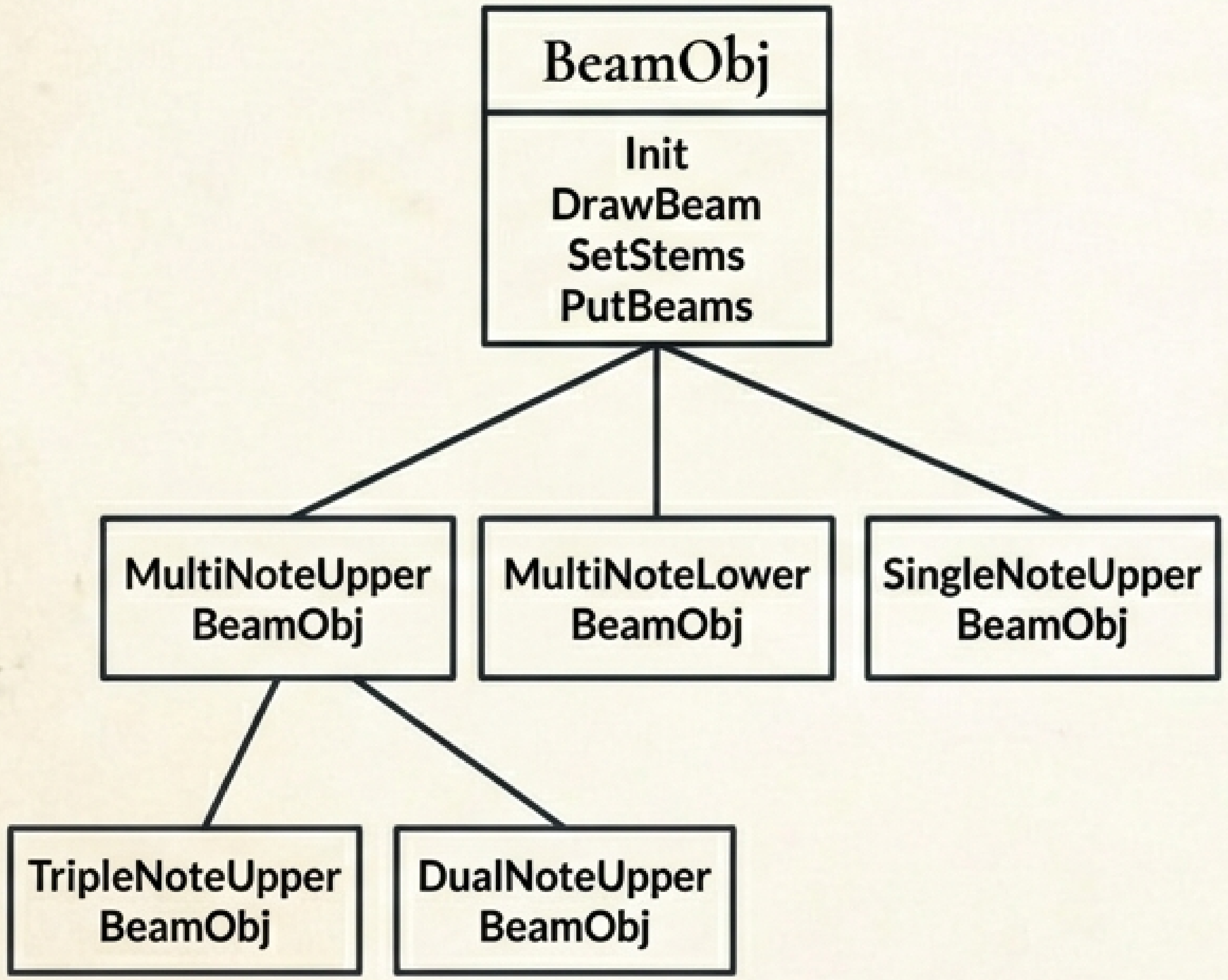
- The rhythmic values of the notes.
- The vertical position (pitch) of the first and last notes in the group.
- The number of notes in the group.
- The overall melodic contour.

A simple line-drawing algorithm is insufficient.

The Solution: A dedicated, object-oriented beaming engine (`MuBeam`). It encapsulates the complex logic and geometry required for perfect notation, treating each beamed group as an object to be rendered.

Inside the Beaming Engine: An Object-Oriented Solution

Key Idea: Instead of one monolithic procedure, `MuBeam` uses a hierarchy of classes, each tailored to a specific beaming case. This makes the logic clear and extensible.



```
// from BeamObj.AddSlopeBonus
procedure BeamObj.AddSlopeBonus(Sign : SignType);
var
  Slope : byte;
begin
  Slope := Abs(Ord(Buffer^.HighestNoteHead^.NoteRec^.GetHeight) -
               Ord(Buffer^.LowestNoteHead^.NoteRec^.GetHeight));
  case Slope of
    0    : Buffer^.ChangeStems(Sign * StemLength div 2);
    1..3 : Buffer^.ChangeStems(Sign * StemLength div 3);
  end;
end;
```

The engine even adjusts stem lengths based on the melodic slope (`AddSlopeBonus`) and position on the staff (`AddHeightBonus`) for optimal aesthetics.

NotebookLM

From Parchment to Performance

A musical score is more than just a visual document; it is a set of instructions for performance. The architecture now shifts focus from static representation to dynamic sound production.

The Scriptorium has produced a perfect score. Now, the Orchestra brings it to life. This requires two critical components:

1. **A Conductor’s Clock**: A precise, real-time timing mechanism.
2. **The Instruments**: A sound synthesis engine to create the notes.

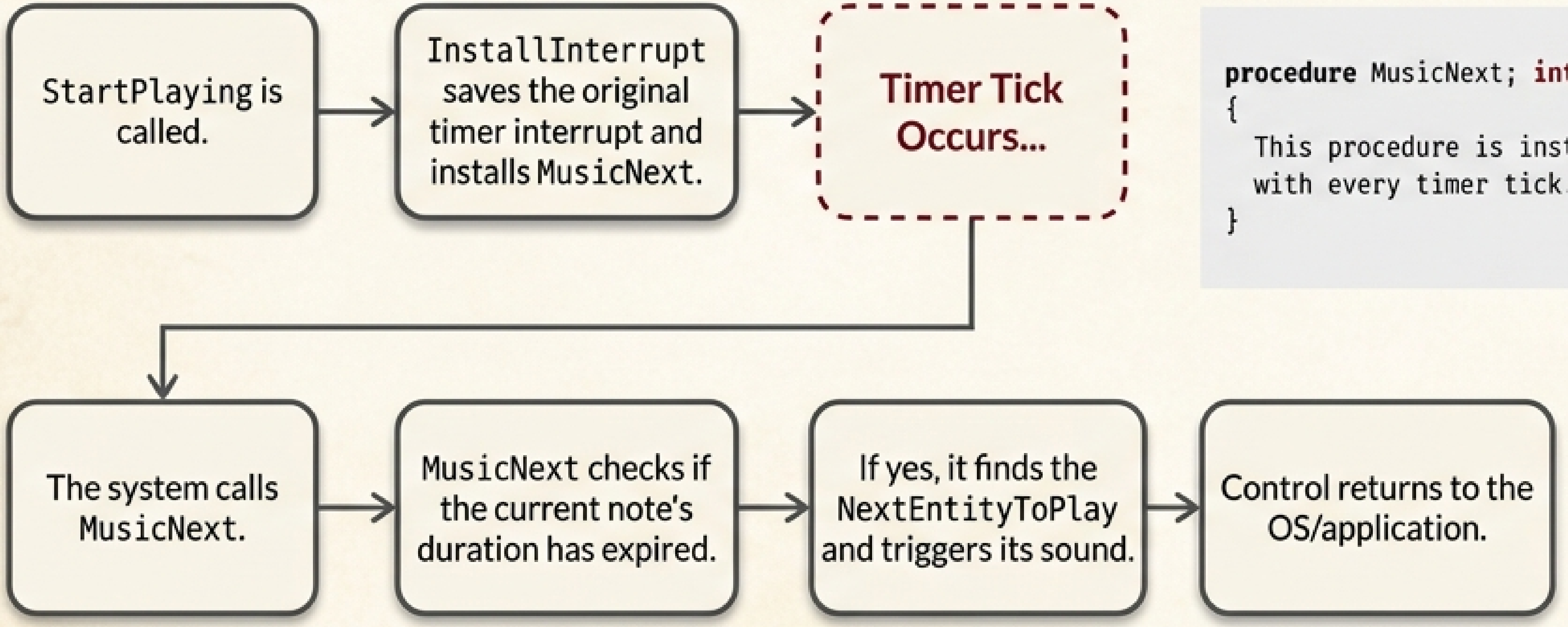
The Metronome's Heartbeat: Real-Time Interrupt-Driven Playback

The Challenge

How do you play music with precise timing in a single-threaded environment without freezing the user interface? Polling loops are inefficient and inaccurate.

The Solution

`MuBckGrd` installs a custom procedure directly into the system's timer interrupt vector. This 'MusicNext' routine runs with every timer tick, completely independent of the main program loop.



StartPlaying is called.
InstallInterrupt saves the original timer interrupt and installs MusicNext.
Timer Tick Occurs...
The system calls MusicNext.
MusicNext checks if the current note's duration has expired.
If yes, it finds the NextEntityToPlay and triggers its sound.
Control returns to the OS/application.

```
procedure MusicNext; interrupt;
{
  This procedure is installed into the timer interrupt, and therefore runs
  with every timer tick. If HereEntity is nil, the music has stopped.
}
```

Giving Music a Voice: The FM Synthesis Engine

Key Idea Text: The MuBlaster unit provides a direct interface to the AdLib/SoundBlaster FM synthesis chip, allowing the software to act as a versatile synthesizer.

How it Works Text: The unit defines instruments by specifying the low-level parameters of their 'Operators' (oscillators). This includes attack, decay, sustain, and release, which shape the sound's envelope over time.

The Instrument Definition

```
FmInstrumentType = record
  SoundCharacteristic : OperatorType;
  OutputLevel : OperatorType;
  AttackDecay : OperatorType;
  SustainRelease : OperatorType;
  WaveSelect : OperatorType;
  Feedback : FeedbackType;
end;
```

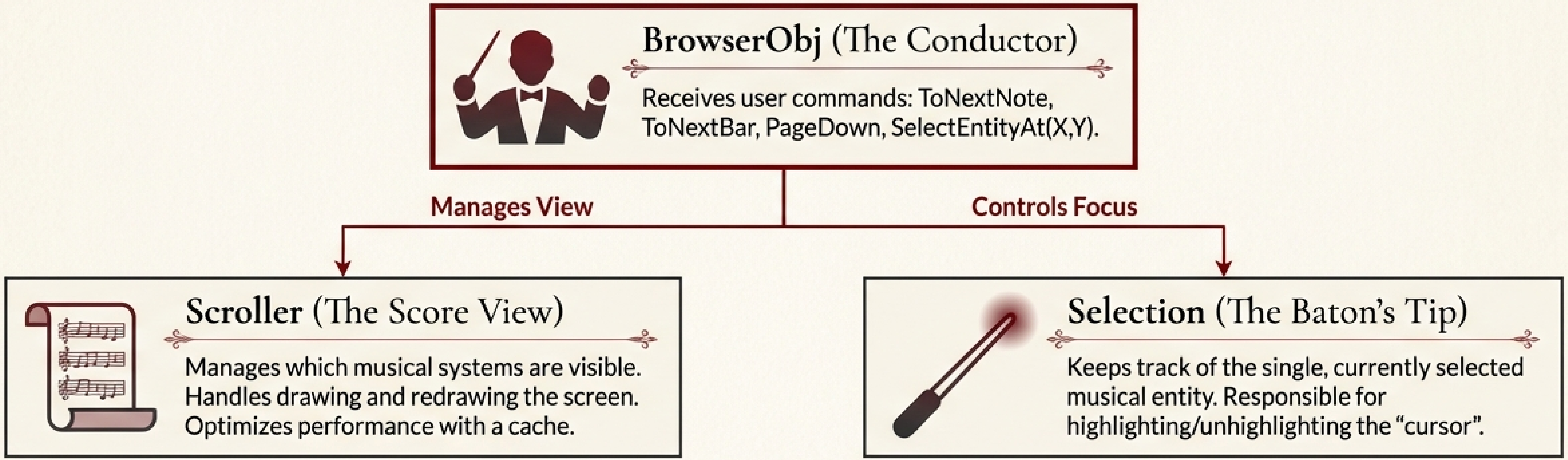
An Example Instrument

```
HoboInstr : FmInstrumentType =
  (SoundCharacteristic :(Op1 : 0; Op2 : 8);
   OutputLevel : (Op1 : 0; Op2 : 0);
   OutputLevel : (Op1 : 0; Op2 : 0);
   AttackDecay : (Op1 : 0; Op2 : 241);
   ...);
```

By changing these byte values, the system can produce a wide range of timbres, from an organ to a hobo.

The Conductor's Podium: Navigating the Score

Key Idea Text: The MuBrowser object is the user's primary tool for interacting with the music. It manages the complex relationship between the data (Piece), what's visible on screen (Scroller), and the current point of focus (Selection).



BrowserObj (The Conductor)

Receives user commands: ToNextNote, ToNextBar, PageDown, SelectEntityAt(X,Y).

Scroller (The Score View)

Manages which musical systems are visible. Handles drawing and redrawing the screen. Optimizes performance with a cache.

Selection (The Baton's Tip)

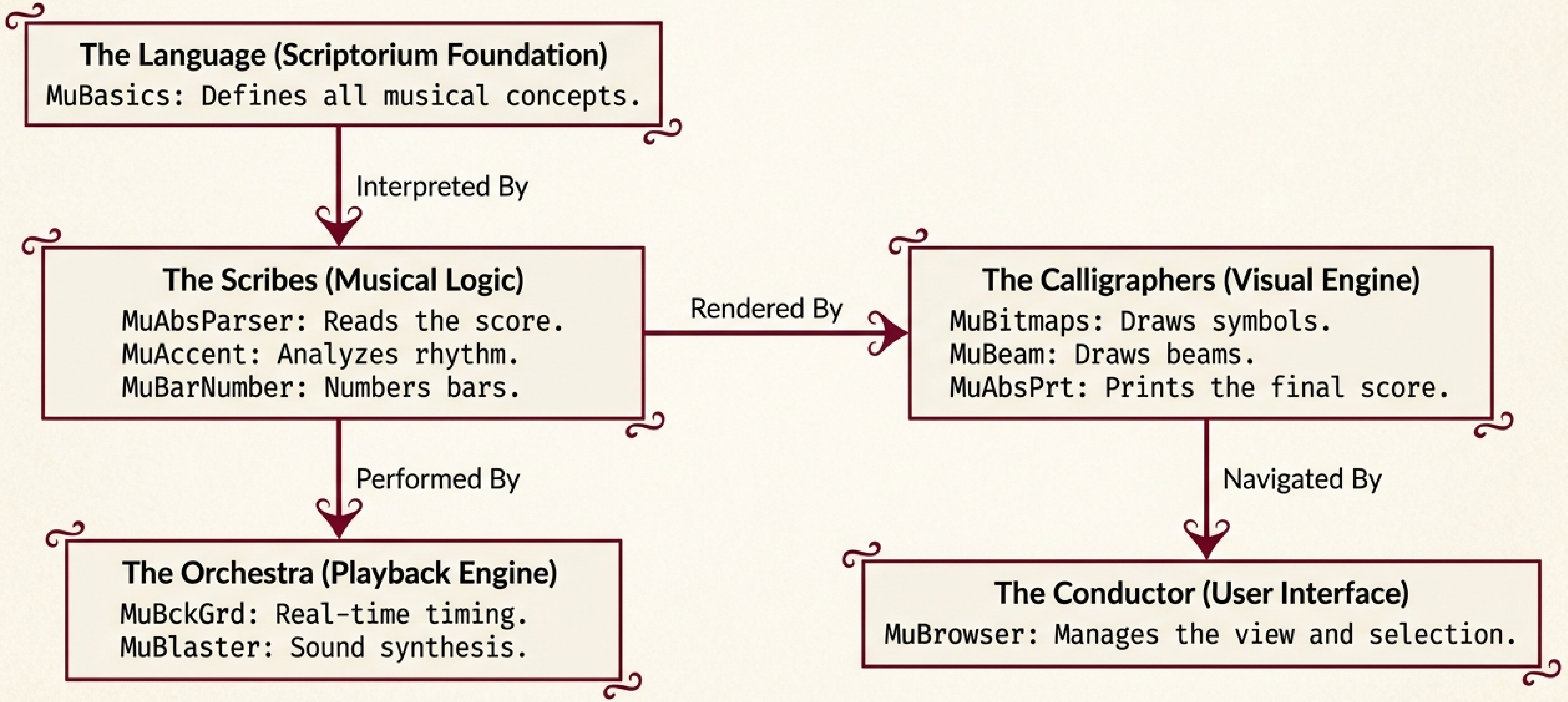
Keeps track of the single, currently selected musical entity. Responsible for highlighting/unhighlighting the "cursor".

This separation of concerns allows for powerful features:

- Point-and-click selection of any note.
- Keyboard navigation (entity-by-entity, bar-by-bar).
- Smooth scrolling and paging through multi-page scores.
- Intelligent screen updates that only redraw what's necessary.

The Complete Composition: A System Overview

Key idea: The system's power comes from the clear separation of concerns and the logical flow of data, from abstract language to visual representation to audible performance.


The Language (Scriptorium Foundation)
MuBasics: Defines all musical concepts.
Interpreted By
The Scribes (Musical Logic)
MuAbsParser: Reads the score.
MuAccent: Analyzes rhythm.
MuBarNumber: Numbers bars.
Rendered By
The Calligraphers (Visual Engine)
MuBitmaps: Draws symbols.
MuBeam: Draws beams.
MuAbsPrt: Prints the final score.
Performed By
The Orchestra (Playback Engine)
MuBckGrd: Real-time timing.
MuBlaster: Sound synthesis.
Navigated By
The Conductor (User Interface)
MuBrowser: Manages the view and selection.

An Architecture of Elegance

This system demonstrates more than just technical competence. It showcases a deep respect for the domain of music, resulting in an architecture that is not only functional but also logical, extensible, and elegant.

1. Domain-First Abstraction: The core data structures in `MuBasics` accurately model music theory, providing a solid foundation for all other logic.

2. Separation of Concerns: The clear distinction between musical logic (accentuation), visual representation (beaming), and performance (playback) creates a modular and maintainable system.

3. Solving Hard Problems Well: Complex challenges like real-time playback and aesthetic beaming are not sidestepped but are met with dedicated, well-crafted, and robust solutions.