Module Interface Specification for ScoreGen

Team #7, Tune Goons
Emily Perica
Ian Algenio
Jackson Lippert
Mark Kogan

April 3, 2025

1 Revision History

Date	Version	Notes
2025-01-17	1.0	Initial version
2025-03-24	1.1	Updated contents of module Uses sections
2025-04-03	1.2	Addressed TA feedback

2 Symbols, Abbreviations and Acronyms

- **MG**: Module Guide
- **M1**: Hardware-Hiding Module
- **M2**: User Interface Module
- **M3**: Score Generation Module
- **M4**: Raw Signal Processing
- **M5**: Audio Feature Extraction (note, key, sig, etc.)
- **M6**: File Format Conversions Module
- **M7**: Audio Recording and Playback Module
- **SRS**: System Requirements Specifications

Contents

3 Introduction

This document details the Module Interface Specifications for ScoreGen. A service designed to transcribe user-recorded musical compositions into accurate sheet music by determining pitch, duration, tempo, and more advanced musical features. The service also aims to provide a user interface to use and interact with the product. Complementary documents include the SRS and MG documents. The full documentation can be found on GitHub.

4 Notation

The structure of the MIS for modules comes from Hoffman and Strooper (1995), with the addition that template modules have been adapted from Ghezzi et al. (2003). The mathematical notation comes from Chapter 3 of Hoffman and Strooper (1995). For instance, the symbol := is used for a multiple assignment statement and conditional rules follow the form $(c_1 \Rightarrow r_1|c_2 \Rightarrow r_2|...|c_n \Rightarrow r_n)$.

The following table summarizes the primitive data types used by ScoreGen.

Data Type	Notation	Description
character	char	a single symbol or digit
integer	$\mathbb{Z}$	a number without a fractional component in (-∞, ∞)
natural number	$\mathbb{N}$	a number without a fractional component in [1, ∞)
real	$\mathbb{R}$	any number in (-∞, ∞)
float	Note	a musical pitch, defined by a frequency in Hz
float	Duration	the length of a note, measured in beats or fractions thereof
float	Rest	a period of silence, defined by a duration
natural number	Tempo	the speed of the music, measured in beats per minute (BPM) as a natural number
float	Dynamic	the volume or intensity of a note as amplitude of a signal, e.g., piano (soft) or forte (loud)

The specification of ScoreGen uses some derived data types: sequences, strings, and tuples. Sequences are lists filled with elements of the same data type. Strings are sequences of characters. Tuples contain a list of values, potentially of different types. In addition, ScoreGen uses functions, which are defined by the data types of their inputs and outputs. Local functions are described by giving their type signature followed by their specification.

5 Module Decomposition

The following table is taken directly from the Module Guide document for this project.

Level 1	**Level 2**
Hardware-Hiding Module	-
Behaviour-Hiding Module	User Interface Module Score Generation Module File Format Conversion Module
Software Decision Module	Raw Signal Processing Module Audio Feature Extraction Module Audio Recording and Playback Module

Table 1: Module Hierarchy

6 MIS of Hardware-Hiding Module

6.1 Module

Hardware-Hiding Module

6.2 Uses

N/A (Hardware-Hiding Module Does not use any other modules.)

6.3 Syntax

6.3.1 Exported Constants

- `DEFAULT_MICROPHONE`
- `DEFAULT_AUDIO_OUTPUT`

6.3.2 Exported Access Programs

Name	Input	Output	Exceptions
initializeMicrophone	None	None	InitializationError
initializeAudioOutput	None	None	InitializationError
readMicrophoneBuffer	None	rawAudioData	ReadError
sendToAudioOutput	audioData	None	PlaybackError

6.4 Semantics

6.4.1 State Variables

- **microphoneState:** A state variable with domain {`inactive`, `active`}.
- **audioOutputState:** A state variable with domain {`inactive`, `active`}.

6.4.2 Environment Variables

- **hardwareDriverLibrary:** The software library that provides low-level driver interfaces for the audio hardware.
- **deviceConfig:** A configuration structure containing parameters and settings for the audio devices.

6.4.3 Assumptions

1. The system is equipped with functional audio input hardware (e.g., microphone) and audio output hardware (e.g., speakers or headphones).

2. All necessary drivers are installed, properly configured, and accessible via the `hardwareDriverLibrar`

6.4.4 Access Routine Semantics

`initializeMicrophone()`:
Precondition: Audio input hardware is present and accessible through the `hardwareDriverLibrary`.
Transition: The state variable `microphoneState` is set to `"active"`.
Postcondition: Upon successful execution, `microphoneState` = `"active"`.
Exception: Raises `InitializationError` if the audio input device fails to initialize.

`sendToAudioOutput(audioData)`:
Precondition: A valid `audioData` stream is provided, and the audio output hardware is available and operational.
Transition: The provided `audioData` is transmitted to the audio output device via the `hardwareDriverLibrary`.
Postcondition: The audio data is played through the audio output device; `audioOutputState` is updated accordingly if applicable.
Exception: Raises `PlaybackError` if the transmission or playback of the audio data fails.

6.4.5 Local Functions

- `detectAvailableHardware()`: Scans and returns a list of available audio hardware devices.

- `configureDeviceSettings(deviceType)`: Retrieves and applies configuration settings specific to the provided `deviceType`.

7 MIS of User Interface Module

7.1 Module

User Interface Module

7.2 Uses

Hardware-Hiding Module (M1): For accessing microphone and audio output devices.
Score Generation Module (M3): For displaying generated scores, and managine score customization settings.
File Format Conversions Module (M6): For importing and exporting files, and displaying PDFs.

Audio Recording and Playback Module (M7): For managing recoring sessions.

7.3 Syntax

7.3.1 Exported Constants

- `DEFAULT_THEME`
- `MAX_UPLOAD_SIZE`

7.3.2 Exported Access Programs

Name	Input	Output	Exceptions
displayUploadInterface	None	None	RenderError
triggerPlayback	audioData	None	PlaybackError

7.4 Semantics

7.4.1 State Variables

- **currentScreen:** A state variable representing the active UI screen. Its domain includes possible screen identifiers (e.g., `"Home"`, `"record"`, `"View PDF"`, `"View MusicXML"`).

7.4.2 Environment Variables

- **displayDriver:** The interface module responsible for rendering UI elements on the user's device.
- **inputDevices:** A collection of hardware devices (e.g., keyboard, mouse) used for user input.

7.4.3 Assumptions

1. User devices support modern UI rendering and interactive functionalities.
2. User device is connected to the internet for MusicXML sheet music generation.

7.4.4 Access Routine Semantics

`displayInterface()`:
Precondition: The `displayDriver` must be available and operational.
Transition: The state variable `currentScreen` is updated to its respective view.
Postcondition: The interface is rendered on the active display screen.
Exception: Throws `RenderError` if the interface fails to render.

`triggerPlayback(audioData)`:
Precondition: Valid `audioData` must be provided, and the required audio output devices should be available.
Transition: Initiates playback of the provided `audioData`.
Postcondition: The audio data is successfully played via the corresponding output device.
Exception: Throws `PlaybackError` if audio playback fails.

7.4.5 Local Functions

- `processAudio(device, audioData)`: Sends audio data to the backend from the specified `device` and returns the processed data.
- `getScore(filePath)`: Retrieves the score as a MusicXML file from the specified `filePath` and displays it in an svg format.
- `validateUserInput()`: Validates input received from the `inputDevices` and returns a status or error code based on its correctness.
- `renderScreen(screenType)`: Renders the specified screen, identified by `screenType`, using the `displayDriver`.

8 MIS of Score Generation Module

8.1 Module

Score Generation Module

8.2 Uses

Raw Signal Processing Module (M4): For receiving clean audio data.
Audio Feature Extraction Module (M5): For extracting musical features from audio data. (Note, Tempo, Key Signature)

8.3 Syntax

8.3.1 Exported Constants

- `DEFAULT_FONT_STYLE`
- `DEFAULT_PAGE_SIZE`

8.3.2 Exported Access Programs

Name	Input	Output	Exceptions
generateScore	noteSequence	.mxl file	GenerationError
customizeScoreSettings	settings	None	ValidationError

8.4 Semantics

8.4.1 State Variables

- **scoreSettings:** A list defining parameters for musical score generation. Its domain includes default and user-specified settings (e.g. time signature).

8.4.2 Environment Variables

- **fileSystemAccess:** The interface responsible for accessing the file system, enabling the reading and writing of files (including .mxl files).

8.4.3 Assumptions

1. Input note sequences are properly formatted and adhere to the expected musical notation standards.

2. The file system has the appropriate write permissions for saving .mxl files.

8.4.4 Access Routine Semantics

`generateMusicXML(noteSequence)`:
Precondition: A valid `noteSequence` is provided, and `fileSystemAccess` is operational.
Output: An .mxl file representing the musical score corresponding to the provided `noteSequence`.
Postcondition: The generated .musicxml file is successfully stored and accessible via `fileSystemAccess`.
Exception: Throws `GenerationError` if the input `noteSequence` is invalid or if an error occurs during file creation.

`customizeScoreSettings(settings)`:
Precondition: The input `settings` adhere to the expected format for score configuration.
Transition: Updates the state variable `scoreSettings` with the new configuration values provided by `settings`.
Postcondition: The `scoreSettings` reflect the updated parameters.
Exception: Throws `ValidationError` if the provided `settings` do not meet the required validation criteria.

8.4.5 Local Functions

- `validateSettings(settings)`: Checks the provided `settings` against a defined schema or set of rules, returning a validation status or error details.
- `renderPDF(noteSequence, scoreSettings)`: Generates a PDF representation of the musical score using the `noteSequence` and the current `scoreSettings`; primarily used for previewing or documentation purposes.

9 MIS of Raw Signal Processing Module

9.1 Module

Raw Signal Processing Module

9.2 Uses

Hardware-Hiding Module (M1): For accessing microphone and audio output devices.

9.3 Syntax

9.3.1 Exported Constants

- `DEFAULT_SAMPLING_RATE`
- `DEFAULT_FILTER_SETTINGS`

9.3.2 Exported Access Programs

Name	Input	Output	Exceptions
dsp	filePath	noteSequence	GenerationError
preprocess	rawAudioData	filteredAudioData	FilterError

9.4 Semantics

9.4.1 State Variables

- **currentSamplingRate:** A numerical variable representing the active sampling rate (in Hz) for audio processing.
- **filterParameters:** A configuration object defining the parameters for the filtering operation (e.g., cutoff frequencies, filter type).
- **currentAudioBuffer:** A buffer that temporarily stores audio data during processing, with a defined size and format.

9.4.2 Environment Variables

- **None.**

9.4.3 Assumptions

1. The input audio data is provided in a readable and supported format (e.g., WAV).

9.4.4 Access Routine Semantics

' `preprocess(rawAudioData)`:
Precondition: A valid `rawAudioData` is provided and `filterParameters` have been initialized.
Output: Returns filtered audio data with noise removed.
Postcondition: The output audio data retains the original `currentSamplingRate` while noise is reduced based on the defined `filterParameters`.
Exception: Throws `FilterError` if the filtering process fails due to invalid input or processing errors.

9.4.5 Local Functions

- `computeSpectralFeatures(audioData)`: Computes and returns spectral features from the provided `audioData` for further analysis.
- `hanning(audioData)`: Applies the specified filter parameters to the audio data and returns the filtered output using a hanning window for preprocessing ScienceDirect.

10 MIS of Audio Feature Extraction Module

10.1 Module

Audio Feature Extraction Module

10.2 Uses

Hardware-Hiding Module (M1): For accessing microphone and audio output devices.
Raw Signal Processing Module (M4): For receiving filtered audio data.
Score Generation Module (M3): To format extracted features properly

10.3 Syntax

10.3.1 Exported Constants

- `DEFAULT_FEATURE_SET`

- `DEFAULT_WINDOW_SIZE`

10.3.2 Exported Access Programs

Name	Input	Output	Exceptions
dsp	filePath	featureSet	GenerationError

10.4 Semantics

10.4.1 State Variables

- `featureSettings`

10.4.2 Environment Variables

- None

10.4.3 Assumptions

- Input audio has been processed by the Raw Signal Processing Module.

10.4.4 Access Routine Semantics

`dsp(filePath)`:
Precondition: Valid `filePath` is provided, and `fileSystemAccess` is operational.
Output:
Audio features such as pitch, tempo, and dynamics or all notes played. **Postcondition:** The extracted data is successfully stored and accessible via other functions.
Exception: Throws `GenerationError` if the input `noteSequence` is invalid or if an error occurs during signal processing.

10.4.5 Local Functions

- `getBPM(audioData)`: Analyzes the audio data to determine the beats per minute (BPM) and returns the calculated tempo.
- `detectPitch(audioData)`: Analyzes the audio data to identify the pitch and returns the detected frequency or note name.
- `getKeySignature(noteSequence)`: Analyzes the note sequence to determine the key signature using the Krumhansl-Schmuckler algorithm Temperley (1999).
- `detectOnsets(audioData)`: Identifies the onset times of notes in the audio data and returns a list of timestamps.

11 MIS of File Format Conversions Module

11.1 Module

File Format Conversions Module

11.2 Uses

Hardware-Hiding Module (M1): For accessing microphone and audio output devices.
Score Generation Module (M3): For saving generated scores as musicXML files.

11.3 Syntax

11.3.1 Exported Constants

- `SUPPORTED_IMPORT_FORMATS`
- `SUPPORTED_EXPORT_FORMATS`

11.3.2 Exported Access Programs

Name	Input	Output	Exceptions
importFile	filePath, format	WAV	ImportError
exportFile	data, format, filePath	musicXML	ExportError

11.4 Semantics

11.4.1 State Variables

- **None.**

11.4.2 Environment Variables

- **fileSystemAccess:** An interface that facilitates file system operations, including reading from and writing to specified file paths.

11.4.3 Assumptions

1. The specified file path exists and is accessible for import operations.
2. The export destination file path is writable.

11.4.4 Access Routine Semantics

`importFile(filePath, format)`:
Precondition: The provided `filePath` exists, and the `format` is supported for import operations.
Output: Returns the file data extracted from raw audio input contained in the file.
Postcondition: The imported data is correctly formatted as .WAV and available for further processing.
Exception: Throws `ImportError` if the file does not exist, is inaccessible, or if the specified `format` is invalid.

`exportFile(data, format, filePath)`:
Precondition: Valid `data` is provided, the `format` is supported for export, and the destination `filePath` is writable.
Output: Saves the provided `data` as a musicXML file at the given `filePath`.
Postcondition: The file at `filePath` is created or updated with the exported data in the specified format.
Exception: Throws `ExportError` if writing the file fails due to permission issues or other I/O errors.

11.4.5 Local Functions

- `convertToRawAudio(fileData, format)`: Converts the provided `fileData` from the specified `format` into raw audio data.
- `writeToFile(data, format, filePath)`: Writes the provided `data` in the specified `format` to the given `filePath` using the `fileSystemAccess` interface.

12 MIS of Audio Recording and Playback Module

12.1 Module

Audio Recording and Playback Module

12.2 Uses

Hardware-Hiding Module (M1): For accessing microphone and audio output devices.
Raw Signal Processing Module (M4): For filtered recorded audio, and noise reduction.
Audio Feature Extraction Module (M5): For extracting musical features from recorded audio.

12.3 Syntax

12.3.1 Exported Constants

- `DEFAULT_AUDIO_FORMAT`
- `MAX_RECORDING_DURATION`

12.3.2 Exported Access Programs

Name	Input	Output	Exceptions
startRecording	None	None	RecordingError
stopRecording	None	rawAudioData	RecordingError
playAudio	audioData	None	PlaybackError

12.4 Semantics

12.4.1 State Variables

- **isRecording:** A Boolean flag indicating whether audio recording is in progress. Domain: {`false`, `true`}.
- **currentAudioBuffer:** A buffer that accumulates raw audio data captured during a recording session.

12.4.2 Environment Variables

- **microphoneAccess:** An interface for accessing and controlling the system's microphone hardware.
- **speakerOutput:** An interface for managing audio playback through the system's speaker.

12.4.3 Assumptions

1. Both the microphone and speaker hardware are functional and accessible via their respective interfaces.

12.4.4 Access Routine Semantics

`startRecording()`:
Precondition: The `microphoneAccess` interface is operational and no recording is in progress (`isRecording = false`).
Transition: Sets `isRecording` to `true` and initiates audio capture by invoking `captureMicrophoneInput(`
Postcondition: Audio data is being accumulated in `currentAudioBuffer`.
Exception: Throws `RecordingError` if the microphone is unavailable or fails to start recording.

`stopRecording()`:
Precondition: A recording session is active (`isRecording = true`).
Output: Returns the captured audio data stored in `currentAudioBuffer` as raw data.
Transition: Sets `isRecording` to `false` to terminate the recording session.
Postcondition: The recording session is concluded, and the captured audio data is available for further processing.
Exception: Throws `RecordingError` if no recording is in progress when invoked.

`playAudio(audioData)`:
Precondition: Valid `audioData` is provided and the `speakerOutput` interface is operational.
Output: Plays the provided `audioData` through the speaker.
Postcondition: Audio playback is successfully initiated via `sendToSpeaker(audioData)`.
Exception: Throws `PlaybackError` if the playback process fails due to issues with the speaker or the audio data.

12.4.5 Local Functions

- `captureMicrophoneInput()`: Captures audio input from the microphone and stores the data in `currentAudioBuffer`.
- `sendToSpeaker(audioData)`: Transmits the provided `audioData` to the speaker for playback.

References

Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 2003.

Daniel M. Hoffman and Paul A. Strooper. *Software Design, Automated Testing, and Maintenance: A Practical Approach*. International Thomson Computer Press, New York, NY, USA, 1995. URL http://citeseer.ist.psu.edu/428727.html.

ScienceDirect. Hanning window. https://www.sciencedirect.com/topics/engineering/hanning-window#:~:text=The%20Hanning%20window&text=The%20general%20shape%20obviously%20gradually,scallop%20loss%20is%20also%20lower. Accessed: 2025-04-03.

David Temperley. What's key for key? the krumhansl-schmuckler key-finding algorithm reconsidered. http://davidtemperley.com/wp-content/uploads/2015/11/temperley-mp99.pdf, 1999. Accessed: 2025-04-03.

Appendix — Reflection

The purpose of reflection questions is to give you a chance to assess your own learning and that of your group as a whole, and to find ways to improve in the future. Reflection is an important part of the learning process. Reflection is also an essential component of a successful software development process.

Reflections are most interesting and useful when they're honest, even if the stories they tell are imperfect. You will be marked based on your depth of thought and analysis, and not based on the content of the reflections themselves. Thus, for full marks we encourage you to answer openly and honestly and to avoid simply writing "what you think the evaluator wants to hear."

Please answer the following questions. Some questions can be answered on the team level, but where appropriate, each team member should write their own response:

1. What went well while writing this deliverable?

   Emily: Since a lot of the code has already been written (or at least planned out), the process of identifying modules and their behaviours for this deliverable went quickly and made the whole process a lot simpler.

   Mark: This document was helpful in creating a better understanding of the future of ScoreGen at a more practical, rather than conceptual level.

   Ian: Splitting the work and dividing into sub teams helped speed up the process of breaking down the two documents and their sections. This was good for work efficiency. Communication before the deadline of this deliverable was productive and informative.

   Jackson: I think this deliverable was a good step to lay out all of our plans for our code. This will honestly benefit us in our development of the app itself, as having this to look back at will benefit us greatly.

2. What pain points did you experience during this deliverable, and how did you resolve them?

   Emily: Making the traceability matrix between modules and anticipated changes proved to be a bit of a challenge. We were directed to identify anticipated changes that ideally affect only a single module, which forced me to think more critically about each AC and why they are needed.

   Mark: As some modules are in the process of completion it was difficult to define specific terms that hadn't yet been implemented, thus completing the documentation

required a lot of communication and assumptions. It is also highly likely that changes will need to be made in future revisions of this document to align with design choices that occur later during implementation.

Ian: One pain point was deciding how abstract some of the descriptions of the module secrets/services in the module guide. A balance had to be struck between being able to adequately describe something vs explaining too many details/choices. This was easily solved by re-reading the MG template, and by taking a look at previous students' work and how they handle this.

Jackson: This deliverable was due very soon after the winter break, and getting back into the flow of school plus reconnecting with the group and getting everyone on the same page was tough. Additionally, breaking down the modules into their specifics proved tough, as creating detailed information about our implementation at this stage. At the same time, the app is still being developed, which was difficult to do.

3. Which of your design decisions stemmed from speaking to your client(s) or a proxy (e.g. your peers, stakeholders, potential users)? For those that were not, why, and where did they come from?

   The majority of design decisions made up until this point did not stem from speaking to clients. The main decisions made such as the algorithmic choices and which file formats to support were decisions made based on the team's knowledge of signals and systems and human-computer interfacing. These decisions included selecting the Fast Fourier Transform for pitch detection due to its performance and efficiency and supporting widely used file formats for distribution purposes (PDF) and musical representation (musicXML).

4. While creating the design doc, what parts of your other documents (e.g. requirements, hazard analysis, etc), if any, needed to be changed, and why?

   N/A

5. What are the limitations of your solution? Put another way, given unlimited resources, what could you do to make the project better? (LO_ProbSolutions)

   Our project aims to create as strong an ability as possible for non-technical musicians to create highly detailed notation, but there are intricacies that are extremely difficult to extract from audio alone. Features like staccato, crescendo, chords, grace notes,

or tempo changes are difficult to differentiate from variance that occurs from regular human playing. Tackling this issue effectively would probably best be done with very advanced signal processing and personally trained, or fine-tuned machine learning models. Given more time, it would also be helpful to implement advanced options for users to maximise precision. If for example there is a music piece with lower confidence sections, such an area where there is a similar likelihood of a note being a fast-played 16th note, or a grace note, it could be possible for the user to toggle through most likely interpretations with playback to determine the ideal representation of their playing.

6. Give a brief overview of other design solutions you considered. What are the benefits and tradeoffs of those other designs compared with the chosen design? From all the potential options, why did you select the documented design? (LO_Explores)

   We considered several design solutions, including purely rule-based algorithms and advanced signal processing techniques. The rule-based approach would have been easier to implement but lacks the flexibility to interpret complex musical nuances like dynamics and articulation. Advanced signal processing, while more capable of handling these intricacies, would require more computational resources and expertise in the field. After weighing the tradeoffs, we chose a hybrid approach that combines signal processing with rule-based methods. This design provides a balance between accuracy, flexibility, and resource efficiency, ensuring a strong foundation for transcription while leaving room for future refinement.