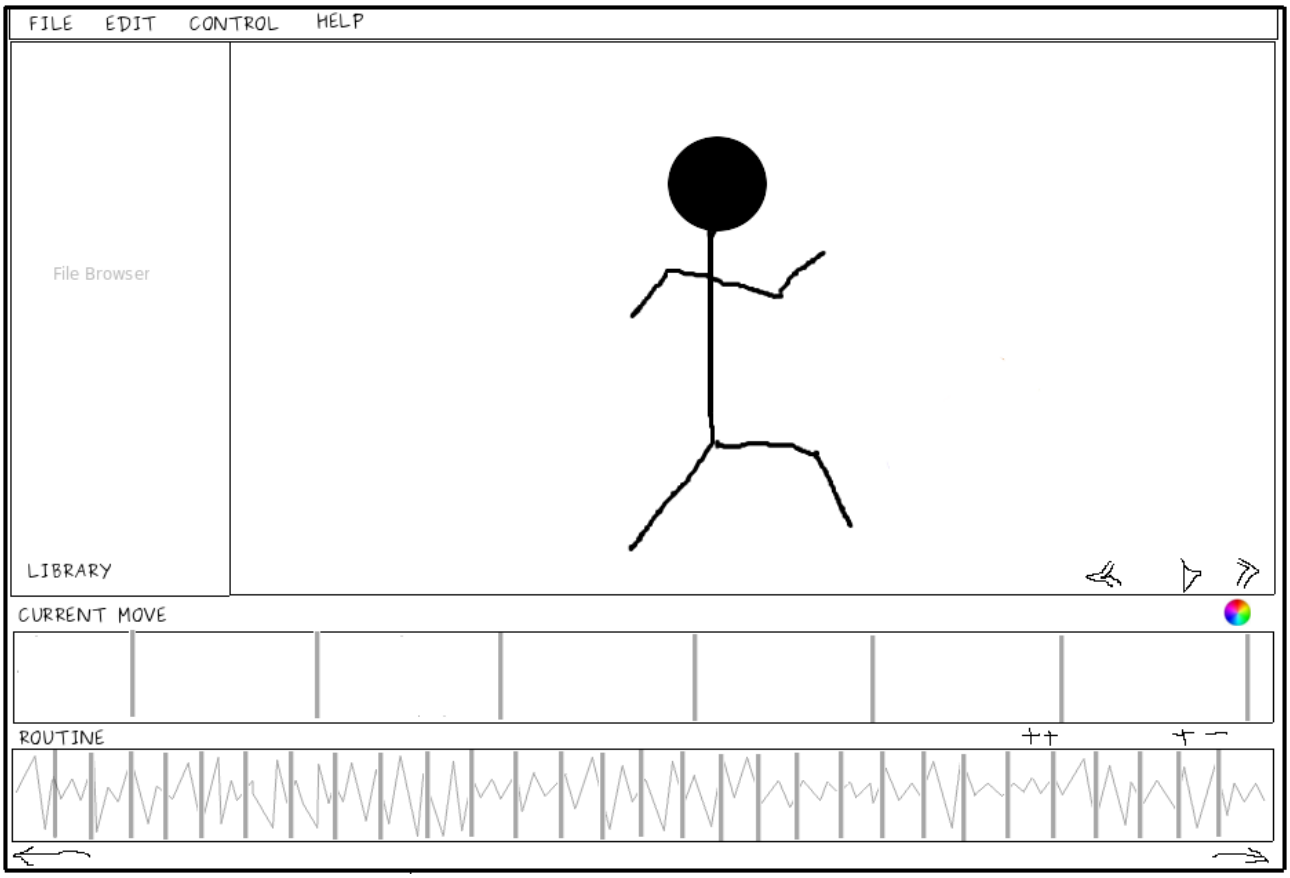
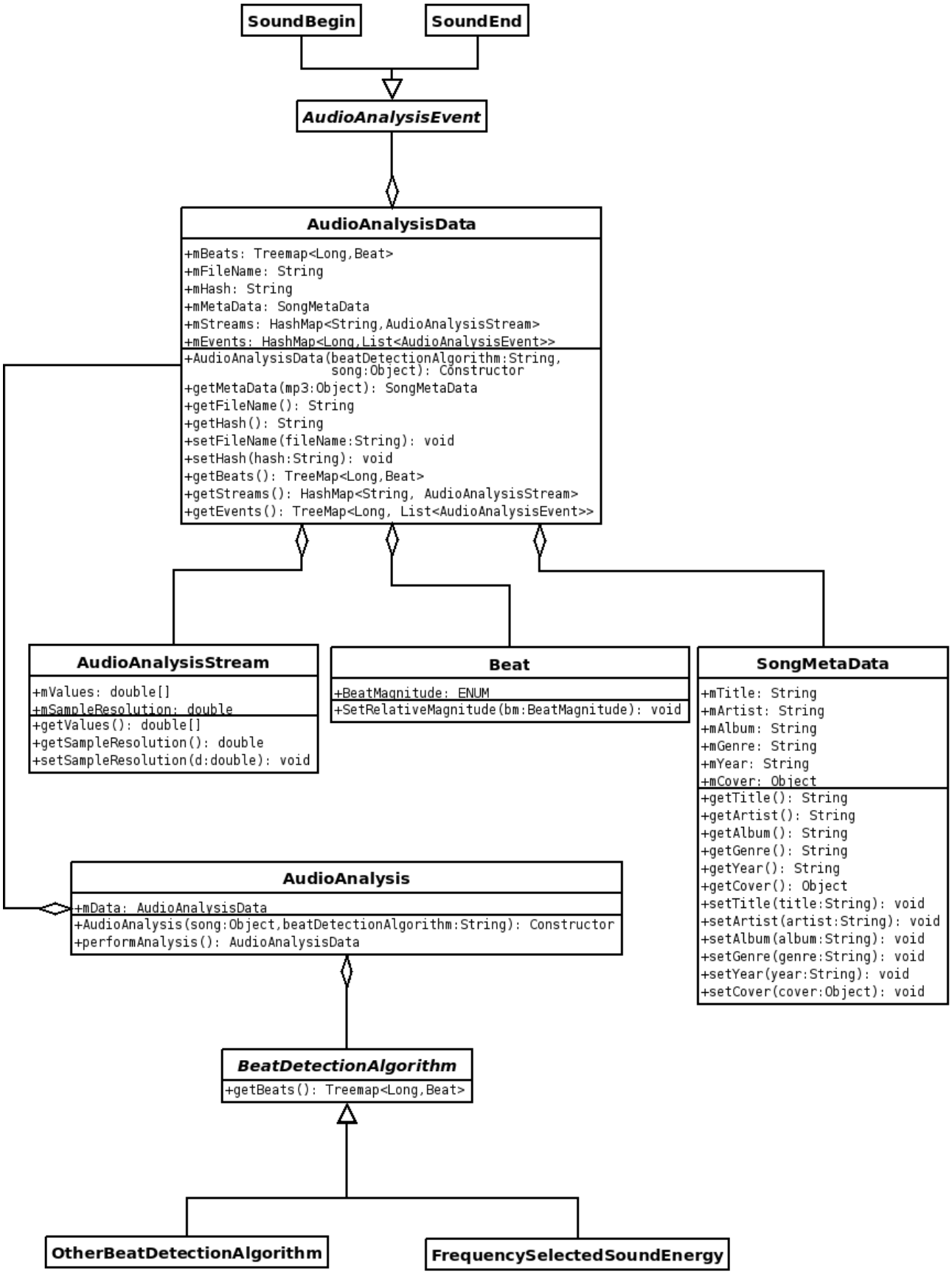
DANCING MAN

Design and Architecture



Contents

Chapter 1: Audio Analysis



1.1. Supported Audio Files : MP3 and OGG Vorbis

The requirements specify that our program must support the MP3 audio format. Unfortunately the Java Sound API is not able to decode MP3 without the help of some extra Java SPI's (Service Provider Interfaces).

MP3SPI is an SPI that extends Java Sound to read, convert and play MP3 sound files. There is also a VorbisSPI that does the same thing but with OGG Vorbis sound files. This means that we can now provide OGG support, seamlessly. The same code that uses the Java Sound API to decode MP3s can be also used to decode OGG Vorbis audio files - No changes are required.

The above method also gives us sufficient low-level access to the decoded PCM data that is required for the audio analysis. The other libraries we tried were too high-level and were not suitable for this.

1.2. Audio Metadata

At the song selection screen, the user of our program will browse their local filesystem for MP3 or OGG Vorbis files. As songs are highlighted, any metadata related to the song will be displayed to them. This metadata can come from a variety of sources.

1.2.1 ID3 tags (MP3 Only)

ID3 is a metadata container used with the MP3 audio file format. This is the first place our program will look for metadata related to an MP3 audio file.

The metadata we will attempt to extract from ID3 tags:

a. Title

b. Artist

c. Album

d. Genre

e. Date

There are a number of Java ID3 libraries available. Some of the ones we tested were JaudioTagger, MyID3 and JID3. We finally settled on MP3SPI because:

a) We are using MP3SPI anyway to provide MP3 Decoding support.

b) It gives us all the metadata listed above and more!

c) No extra libraries for handling tags are required.

d) It worked on hundreds of different MP3 files that we tested it with.

1.2.2 Vorbis Comments (OGG Vorbis only)

ID3v2 tags break formats which are container-based such as Ogg Vorbis, so it uses its own tagging format. This is the first place our program will look for metadata related to an OGG Vorbis audio file.

The metadata we will attempt to extract from Vorbis Comments:

a. Title

b. Artist

c. Album

d. Genre

e. Date

We decided to use VorbisSPI to gather OGG Vorbis metadata for the exact same reasons we chose MP3SPI to gather MP3 metadata.

1.2.3 Metadata obtained from the web.

We are sure to encounter MP3/Ogg Vorbis files that contain little or no metadata. Any information missing from these files will instead be searched for using freely available online music databases.

There were two main databases to choose from: muzicbrainz and FreeDB. We finally settled on musicbrainz because:

a) There are a lot of people spending a lot of effort making sure the data is accurate and consistent.

b) There exists a Java API for accessing the musicbrainz database.

c) We know someone who has used musicbrainz before for collecting song metadata from the web and they were very successful.

1.2.4 Album Covers

If we are able to determine the album that the highlighted song belongs to, its album cover will be displayed along with the textual metadata.

Amazon has a gigantic collection of album covers and Amazon Web Services provides an API to easily access these.

1.3. Beat Detection Algorithms

An important part of the audio analysis process is the detection of the location and magnitude of each beat in the song. There were three main beat detection algorithms that we considered. They are briefly described below, listing the good and bad points of each one.

1.3.1 Simple Sound Energy

Detects sound variations by computing the average sound energy of the signal and comparing it to the instant sound energy. A beat is detected when the energy is superior to a local energy average.

Good:

- The algorithm is very efficient and quick to execute.
- The algorithm is far simpler to implement than the other two.
- Beat detection is very accurate with techno and rap songs.

Bad:

- Doesn't detect beats in in songs with a lot of instruments playing different rhythms simultaneously.
- Beat detection on most songs (anything that is not techno or rap) is very approximate.

1.3.2 Frequency Selected Sound Energy

Detects big sound energy variations in particular frequency sub-bands. Determines on which frequency we have a beat and if it is powerful enough to take it into account.

Good:

- Accurate beat detection for a variety of different music genres.
- Allows us to separate beats by their frequency sub-band.
- Easily adaptable to any kind of signal. Beats can be selected according to many different criteria.

Bad:

- A little more complex to implement compared to the first algorithm. It requires computing the Fast Fourier Transform.

1.3.3 Derivation and Comb Filter

The signal is broken up into frequency bands and each of these is transformed by emphasising the sudden impulses of sound in the song. Combfilter processing is then used to determine the principal BPM rate.

Good:

- The most accurate beat detection for a variety of different music genres.

Bad:

- Much more complex to implement than the other two beat detection algorithms.
- Extremely computationally expensive! Usually it is only executed on a small part of a song.

1.3.4 Conclusion

Our program will implement the second - 'Frequency Selected Sound Energy' algorithm. This is mainly due to the shortcomings of the other two.

The 'Simple Sound Energy' is far too limited in that it only performs well with techno and rap music. We want our users to be able to use a large variety of songs in our dancing man program.

The 'Derivation and Comb Filter' algorithm would have been brilliant if it wasn't such a slow and complex algorithm. We want to analyse entire songs in a reasonable amount of time. The 'Derivation and Comb Filter' algorithm simply doesn't meet this requirement.

We can take advantage of the fact that the 'Frequency Selected Sound Energy' is easily adaptable. Depending on the song genre, we can slightly modify the algorithm to perform optimally for songs of that genre.

We will make use of the strategy pattern so that the program can be easily extended to use different beat detection algorithms. Each beat detection algorithm will be encapsulated in its own class and will implement the 'BeatDetectionAlgorithm' interface. By allowing beat detection algorithms to be pluggable we can dynamically choose the best algorithm for a particular input.

1.4. Other Audio Analysis Outputs

1.4.1 Tempo aka. BPM

Once we have the location of each beat in the song we can easily calculate the overall tempo. We can also calculate localized tempos, ie. the average tempo for every consecutive XX second segment.

1.4.2 Song Hash

A hash that (almost) uniquely identifies the song in question. This will be represented as a string of hexadecimal digits. We will use Java's MessageDigest class with the SHA algorithm to generate the hash value.

The song's hash will be stored in the routine file so that next time a user loads the routine and selects a song, the program can use the hash to verify that the chosen song is identical to the one that was used to create the routine.

1.4.3 Streams

Streams are used to store continuous audio analysis data that changes throughout the song. Each

different statistic we are measuring has its own stream, and at each point in time will have a particular value. Each stream also has its own sampling resolution (some data we want to sample more frequently than others).

Streams can be used to record things such as:

- Localised BPM measurements (described above)
- Volume
- Pitch

1.4.4 Events

These on the other hand store non-continuous data. Unlike with streams, all of the different events are combined into the same data structure. Since two or more events could occur at the same time in a song, we map each time to a list of events.

Events can be used to record things such as:

- The position of a high part in the song. A high part of the song is a section with significantly higher amplitude/volume. This quiet often indicates the position of the chorus.

- When the music actually starts and finishes playing. Some songs have periods of silence at the beginning and/or end. The program may be able to detect these silent periods and create events where the music actually starts and finishes playing.

1.5. Audio Playback

1.5.1 Small Audioclips

We have created an 'AudioClip' class to handle the loading and playback of small audio files (usually under five seconds in length). It was written due to drawbacks of Java Sound:

1. Java Sound has a limit to the number of Lines you can have open at the same time (usually 32). Clips are lines so this means we only have a limited number of sounds.

2. The same clip cannot be played simultaneously. (eg: two sirens going off at the same time).

In our program any number of sounds can be loaded, and several copies of each sound can be played simultaneously.

1.5.2 Main Song Associated with the Routine

We will create a music player that supports the following operations:

a) Load a new song from a File

- If any songs are currently playing they will first be stopped.
- The chosen song will be loaded into the player, ready for playback.

b) Play the currently loaded song

- The song that has been loaded into the music player will start playing in a new thread.
- This could be a newly loaded song or a song that has been stopped.

c) Pause the currently playing song.

- This will only have an effect when a song is loaded and is currently playing.

d) Resume the currently paused song.

- This will only have an effect when a song is loaded and is currently paused.
- Playback will resume from the song position at which it was paused.

e) Stop the current playing song.

- This will stop the currently playing song and free any resources associated with it.
- After stopping, if the user chooses to play the song again, it will start playback from the beginning of the song.

f) Seeking to a new position in the current song.

- If the song was paused, then the song will remain paused after seeking to the new position in the song.
- If the song was playing, then after seeking the song will continue to play from the new position in the song.

g) Return the current song position.

- This will be continuously polled by the GUI backend so that it can update the slider bar to reflect the new song position.

1.5.2.1 Events

SONG_END - This event indicates to the GUI that the end of the song has been reached and is now stopped.

There may be a very small delay between when the user clicks a music player control on the GUI and when the chosen action actually takes place. This might be because the music playback thread is busy doing another task or waiting to get use of the CPU. If these delays becomes an issue, some other events (such as STOP, PAUSE etc.) may need to be generated to ensure that the GUI music player controls are synchronized with the player state.

1.6. Interface between Audio Analysis and the Routine Editor.

This is a core component because it provides a bridge between two important and distinct modules in our dancing man system. A well defined interface is essential because it ensures low coupling. The different modules need not be concerned with the other modules internal implementation. They only deal with each other through this interface.

1.6.1 Why we chose a TreeMap to represent beats.

TreeMap<Long,Beat>.

a) It provides a mapping between a time in the song and the beat at that point.

b) It orders the beats by their location in the song.

c Can easily find a beat nearest to a given location.

d) Can easily access beats that are nearby to a given beat.

Chapter 2: GUI Frontend

--Strategy--

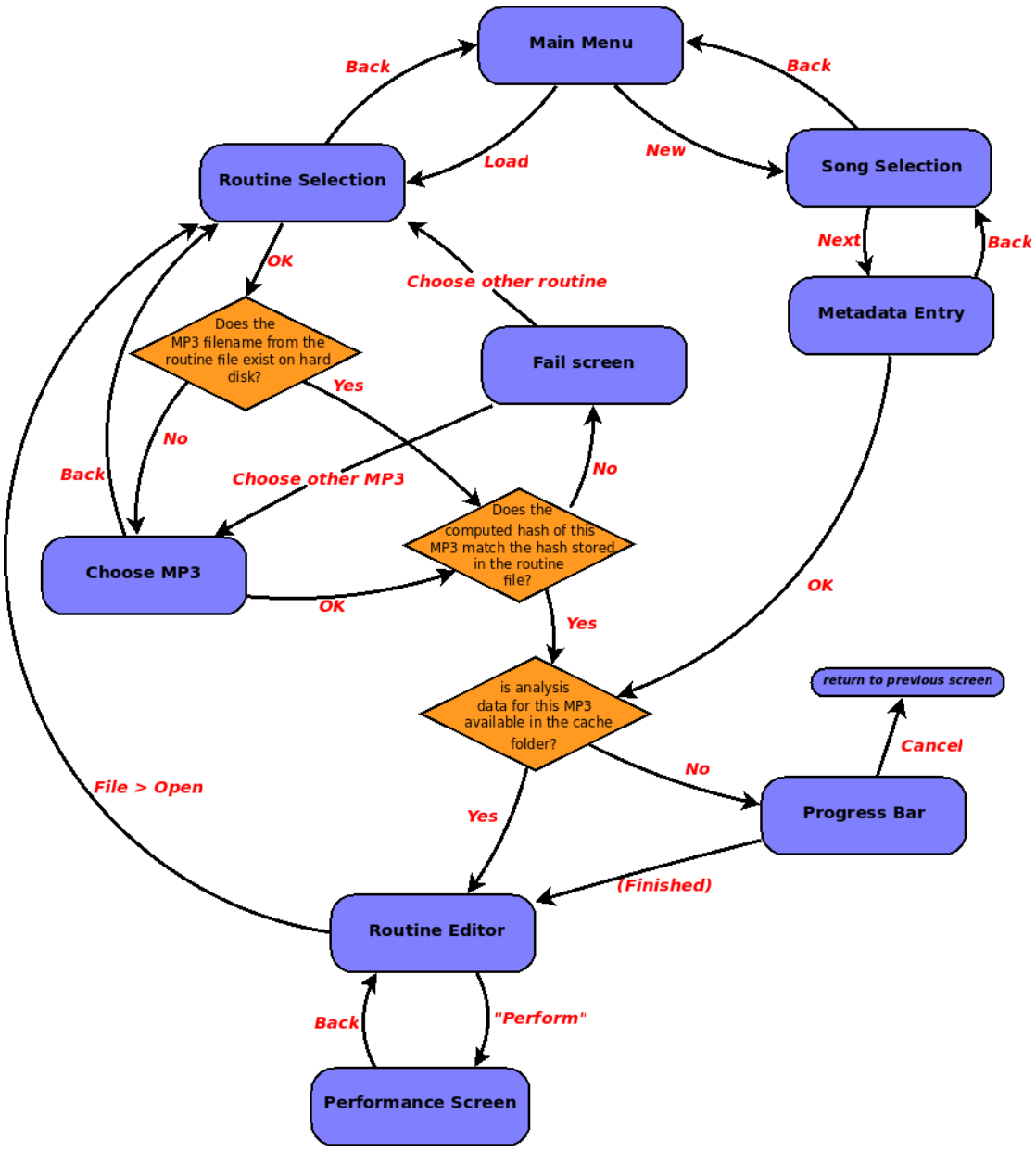
There are to be a series of different stages (i.e. screens) that the user goes through to make themselves a dance routine. On opening the program the user is first presented with the main menu screen. They will then go through a number of different screens (the order depending on each option they choose in the screens) before getting to the main part of the program; the routine editor. The user will spend time modifying their routine to make the stick man dance to the music. Once the routine is finished they can then view it in performace mode (i.e. in the performance screen).

--Theme--

The GUI for the dancing man project has an overall theme for each of the screens. Each screen is to have the effect of looking like it has been drawn with pen and paper, similar to a comic book and what is drawn on 'http://xkcd.com/'. Colour schemes for the project are to remain very simple. This will add to the comic-like effect. Mainly each of the screens are drawn in black and white but small splashes of colour will also be used to make the screens more interesting and to add dramatic effect. Using this comic-like theme will also work well with the 2D dancing stick man. The main font-family that will be used in each screen is Pusia. This is an ideal font for the theme that will look good in a variety of different styles. Purisa is already available in the java GraphicsEnvironment so this make things easy.

--Navigation--

Refer to ScreensFlowchart.png for a decription on how the user will navigate around the dancing man program. This will also show how each screen is linked to one another.



--Screen Components--

Each screen contains a number of components:

-Main Menu:

Description: The Screen that is viewed on opening the program.

Components:

Labels: The Dancing Man!

Main menu

Buttons: New

Load

Quit

Graphic/Animation: A picture or preview of a previously made dancing man move.

-Song Selection:

Description: The user has chosen to make a new routine and now needs to choose a song to make a this new routine to.

Components:

Labels: Song Selection...

Choose a song from your computer to make a routine to.

Song Information (Title, Artist, Album, Genre, Year, Album Art)

Dynamic Labels:

Song information labels will depend on the ID3 tags of the currently selected song in the file browser.

Buttons: Next

Back

File Browser: An on-screen file browser will be displayed.

-MetaData Entry:

Description: A screen for the user to add a little bit of information about the routine they are about to create.

Components:

Labels: Routine Information...

Please enter details about your routine

Buttons: OK

Back

Input feilds: Routine Name

Creators Name

-Progress Bar:

Description: A loading bar to inform the user on the progress of the audio analysis.

Components:

Labels: Audio analysis in progress...

Graphical progress bar: updated to represent how much audio analysis is

done.

Buttons: Cancel

-Routine Selection:

Description: The user has chosen to load an existing routine and now needs to search their hard drive for the one they want.

Components:

Labels: Routine Selection..

Choose a saved routine file from disk to load

Routine Information (Routine Name, Routine Creator)

Routines song information (Title, Artist, Album, Genre, Year, Album Art)

Dynamic Labels: Routine Info and routine song info will depend on the currently selected routine in the file browser.

Buttons: OK

Back

File Browser: An on-screen file browser will be displayed.

-Choose MP3:

Description: The MP3 used to create the selected routine couldn't be found on disk so the user needs to find it manually.

Components:

Labels: The MP3 file this routine was created for was not found. Please select an MP3 that can be used in its place.

Song Information (Title, Artist, Album, Genre, Year, Album Art)

Dynamic Labels:

Song information labels will depend on the ID3 tags of the currently selected song in the file browser.

Buttons: Next

Back

File Browser: An on-screen file browser will be displayed.

-Fail Screen:

Description: The MP3 the user has selected doesn't match (by hash) the MP3 that that was used to create this routine.

So the user now needs to either find another song that does match or choose an entierly different routine.

Components:

Labels: Error! The selected MP3 file is not the same MP3 that was used to create this routine.

What would you like to do?

Butons: Choose another routine

Choose another song

-Performance Screen:

Description: A large screen to watch the routine of the stick man. (ususaly once the entire routine is completed)

Components:

A large screen to view the dancing mans routine.

Buttons: Back

Performance playback controls: Play

Pause

Stop

Seek

Progress Bar: to display how far through the routine is.

-Routine Editor:

Description: This is the main part of the program. The user will use this screen to do all of the editing of the

stick man and create their own moves.

Components:

For a detailed description of the components refer to section 5: Routine Editor.

The analysis files are stored in the program's own folder, in an analysis subdirectory. The files are named according to the hash of the MP3 file that they are the analysis of.

When creating a new routine

1. Present user with song selection screen; user selects an mp3 to use.

2. Present user with metadata entry screen; they enter in their name and the name of the dance they are about to make.

3. Calculate the hash of the selected mp3 file.

4a. If an analysis file exists with that hash as the filename, load the analysis from it.

4b. Otherwise, do the audio analysis (show a progress bar), saving the outcome to the analysis directory.

5. Load up the editor, giving it the analysis data (including mp3's id3 metadata), user's metadata, hash of mp3 and filename of mp3.

When loading a routine

1. Present user with routine selection screen; user selects a routine file to load.

2. Read 'mp3 path', 'mp3 filename' and 'mp3 hash' from routine file.

3a. If the folder the the routine file is in contains a file called 'mp3 filename', 'select' this file and go to step 4.

3b. Otherwise, if the folder specified in 'mp3 path' contains a file called 'mp3 filename', 'select' this file and go to step 4.

3c. Otherwise, present user with song selection screen; user 'select's an mp3 file to use.

4. Calculate the hash of the mp3 file that has been 'select'ed.

5a. If the calculated hash is different from 'mp3 hash',

i. Ask the user if they want to ignore this discrepancy.

iia. If they do, go to step 6.

iib. Otherwise, go to step 3c.

6a. If an analysis file exists with that hash as the filename, load the analysis from it.

6b. Otherwise, do the audio analysis (show a progress bar), saving the outcome to the analysis directory.

7. Load up the editor, giving it the analysis data (including mp3's id3 metadata) and routine data.

When saving a routine

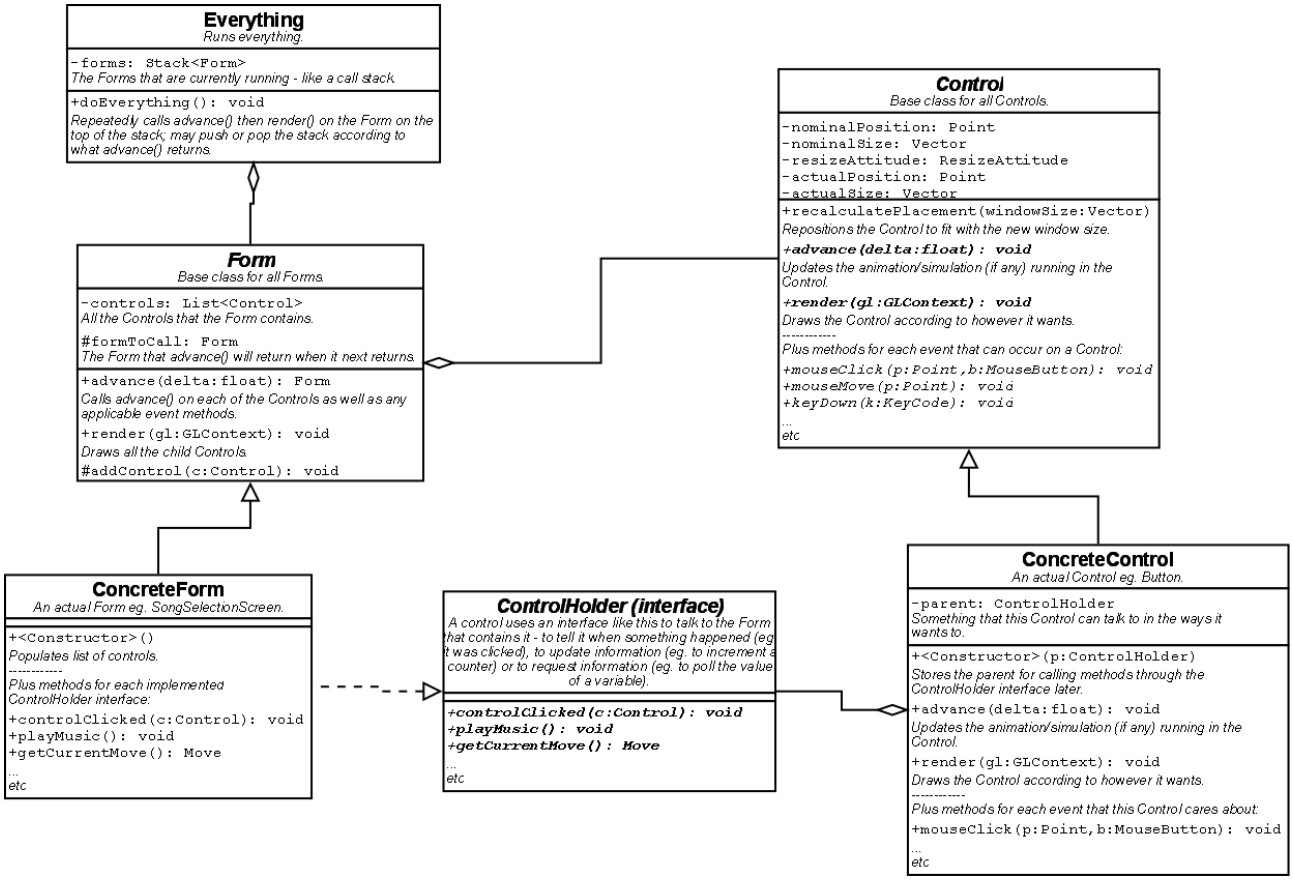
1. Present user with save dialog box; user selects a directory and filename to save to.

2. Save all the routine, move and keyframe data, the user's metadata, the title and artist of the mp3, the hash of the mp3 and the path and filename of the mp3 to the file.

Chapter 3: GUI Backend

This section refers to the package comp314.guibackend. Refer to the javadocs of the classes in that package for more information.

The GUI Backend handles all the non-content specific GUI code. The GUI of the program consists of a series of Forms, each containing a series of Controls, which are drawn and can be interacted with. A Form occupies the entire window - that is, there is exactly one Form being run at a time. The program will do all its rendering using OpenGL, and get all its input using Swing's event system. Once the events from Swing are intercepted and cached, the entire system is single-threaded and polling based.



Forms and navigation between them The Everything class is the top-level class of the program, and is always 'controlling' the execution (in that it contains the main execution loop). The Everything class keeps a stack of Forms. The Form on the top of the stack is the one that is currently being run/displayed - its advance() and render() methods get called each frame. The advance() method of a Form returns an instruction to the Everything class: to keep running the same form, to call a new one (push on the stack), or to return to the previous one (pop off the stack). What the advance() method returns is a protected variable that the concrete Form modifies when it sees fit. This variable

is of type Form, and is null to return, this to stay on the same Form or another Form to call it.

Advancing controls

The advance() method of the Form class calls advance() on each of its child Controls. This is to update any animation or simulation that may be going on in the Control. All advance() methods are passed the amount of time which has elapsed since they were last called.

Controls being sent events

The Everything class gets all the Swing mouse and keyboard events, and puts them into an Input data structure. The current Form then uses this data structure to dispatch events to Controls inside it. As well as calling Control.advance(), Form.advance() calls all appropriate event methods on all appropriate Controls (eg. if the mouse has just been clicked, Form.advance() finds the Control the mouse is over and calls mouseClick() on it).

Rendering controls

The render() method of the Form class simply calls render() on each of its child Controls. Each Control is responsible for drawing itself. All the render() methods are passed a GL context in which to call GL commands. This will be backed up with a reference-counting TextureManager class with static methods to load and use textures, as well a a GraphicsUtilities class with static methods for common sequences of GL commands, like drawing a box or a line.

Differing window sizes and aspect ratios

The program will tolerate any window size/aspect ratio, and will attempt to reposition the Controls on a Form nicely. Each Control specifies its position and size in 'pseudopixels' - which is where it would be if the window size was 1024x768 pixels. It also has a ResizeAttitude, which is used to determine where to put the Control when the window is resized - eg. lock it to the left side of the screen, stretch it vertically etc.

Controls communicating with Forms

If a Control needs to be able to talk to the Form that it holds, the Form will implement an interface which contains methods for all the ways the Control will interact with its parent. The Control will have an instance variable which is supplied in its constructor. When a Control wants to interact with its parent Form it calls the appropriate method in its parent which will do whatever it should do. For example:

A Form contains a Button. The Form implements ClickableHolder, which exposes the Form's controlClicked(Control c) method. When the Button is created the Form passes itself as the parent, which the Button stores as an object of type ClickableHolder. When the Button gets clicked (ie. In its mouseClick() method), it tells the Form about it by calling parent.controlClicked(this). Then the Form's controlClicked(Control c) method will presumably do a switch(c) and away we go.

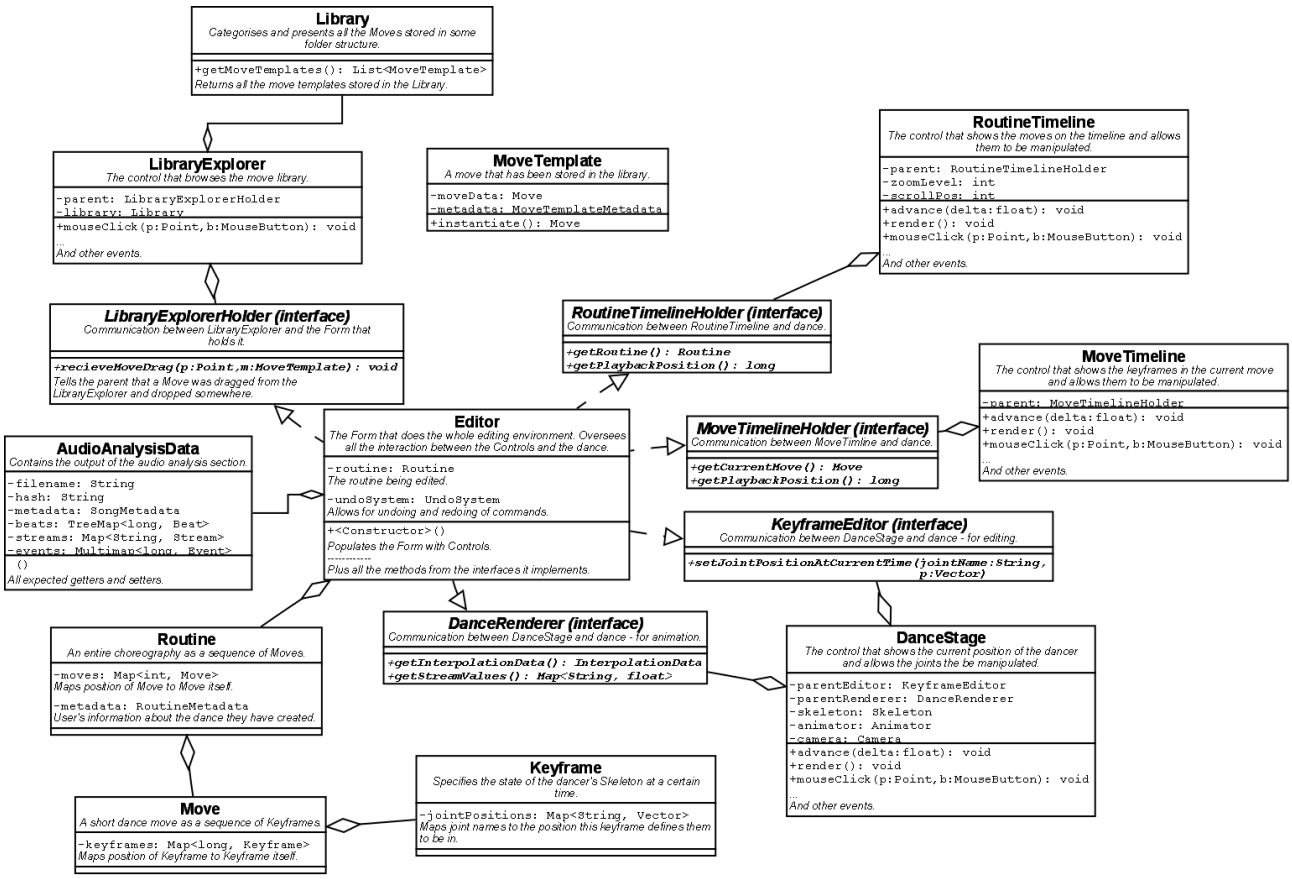
Undo system

The undo system provides an easy way of undoing and redoing actions just like in any other

program. An atomic operation that can be done or undone is represented by an Operation, which will have one subclass for every possible operation. Operation has undo() and redo() methods which can be implemented any way the particular Operation needs. The UndoSystem class keeps two lists of Operations, one of Operations that can be undone (in reverse order of how they were done) and one of Operations that have been undone and can be redone (in reverse order of how they were undone). An undo or redo is then just a bit of a shuffle of these lists, calling a function on the Operation on the front of one of them. There is a CompoundOperation class for grouping together Operations that were done together as one atomic Operation that will be undone together.

Chapter 4: Routine Editor

The routine editor is the main part of the program, where the user creates a dance. The Editor class, a Form, is the main class.



Routine structure

An entire dance is represented by a Routine, which contains a sequence of Moves, each of which contains a sequence of Keyframes.

The Routine also contains some meta-data about the routine, the name and hash of the MP3 file used for the Routine. That sort of stuff is described in Front End. This map means that given a beat index, we can check if that beat has a move on it and retrieve that move. As it is a TreeMap we can also retrieve the first move before and the first move after a beat.

All modifications pass through the Editor

The Editor contains the current structure of the dance. All modifications to the dance come from the Controls (mainly RoutineTimeline, MoveTimeline and DanceStage), which call methods in the Editor which are exposed to them through interfaces (eg. RoutineEditor, MoveEditor and KeyframeEditor). This is an example of the interface system described in GUI Backend. The Controls don't store any data about the object they are editing; these are provided to them through the interfaces as

Editor methods like getRoutine() and getCurrentMove(). Since all the modifications go through the

Editor class, the Editor can keep track of and oversee easily any changes to the dance structure, which allows it to contain an UndoSystem as described in GUI Backend and let all such changes be undoable.

Editor front-end GUI

Following is a basic description of the editor environment.

Menu bar:

Seldom-used functions or functions where the primary method of executing them will be with the mouse or keyboard shortcuts will be in the menus.

Dance Stage:

The current state of the stick figure is shown on the screen. The joints can be moved around to create or modify keyframes and the camera can be moved around.

Buttons:

A few common functions like play and pause will be represented with ever-present buttons on the screen.

Library Explorer:

The library contains pre-built moves that will be able to be dragged on to the routine timeline. The library can be shown and hidden.

Move Timeline:

Shows the keyframes for the current move. Allows the user to manipulate keyframes. The keyframes are shown as boxes in the color that the move has been given.

Routine Timeline:

Shows the moves arranged in the routine. Allows the user to manipulate moves. Can be scrolled and zoomed. Each move on the routine timeline has a color to identify it by which can be changed.

One of the timelines is active at a time - this is the one that has focus for otherwise ambiguous commands. The inactive timeline is shown slightly greyed out. Any action on a specific timeline makes that timeline become the active one.

The position of the playback head completely determines which move is the 'current move' and the position of the dancer etc. Every action that modifies the position of the playback head can thus change the current move. If the playback head is in an empty space on the routine timeline, the move timeline becomes blank and unusable. The playback head is shown in both timelines as a red line.

Pluggable renderers

A Renderer is used by the DanceStage Control to draw the dancer. Multiple Renderers will be able to be implemented, drawing the dancer using different effects etc. This allows for eg. a slightly different Renderer to be used for performance mode than for the Editor, which could include flashy lighting etc.

User interactions

See the separate file RoutineEditorUserInteractions.txt.

Automatic Dance Generation

Initially automatic dance generation was to be a large part of the dancing man system, but we have instead shifted our focus to a fully-featured routine editor that the user can use to create their own routines.

The audio analysis output will be used to generate a very basic dance for the user to use as a template.

Possibilities include:

- The animated stick figure tapping his foot to the beat.
- The animated stick figure nodding his head to the music.