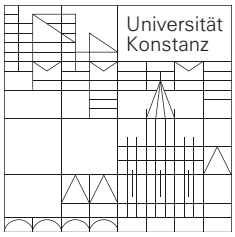
HAIL:
Developing the Human-Audio Interaction Lab

Master Project Paper

by

Sebastian Fichtner



Faculty of Computer- and Information Science

Prof. Dr. Harald Reiterer
Dr. Hans-Christian Jetter

Konstanz, 2014

Abstract

This master project is about the interaction design of music composition software. In the previous master seminar [3], we worked out specific requirements. Here, we report from our practical attempts to satisfy those criteria. This includes the process of domain modeling, sketching and prototype implementation of our Human-Audio Interaction Lab. We propose a relative simple domain model that allows to solve many crucial requirements at one stroke.

Contents

1 Introduction

In this master project, we designed, implemented and visualized a basic domain model of music composition that allows a novel way of interacting with composed audio. We propose a design approach that makes temporal- and voice abstractions available to the user as explicit objects. We demonstrate how a generalization of concepts can lead to a more powerful and, yet, simpler and more elegant interface.

Figure (1) shows a screenshot of our prototype. This project report will explain how we arrived at this interface, what it pictures and what can be done with it.

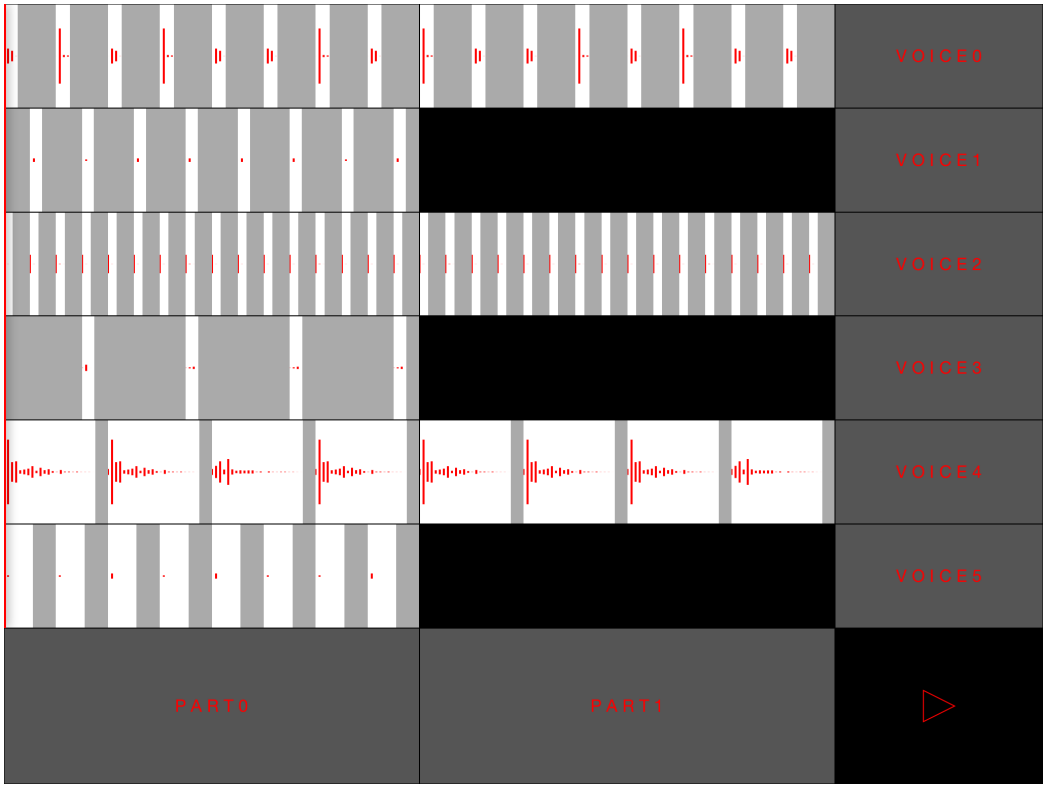


Figure 1: A performance of two parts and six voices in our prototype

We already detailed our general approach of Domain-Driven Interface Design in [3]. But we want to reiterate that, as this approach is based Domain-Driven Design [1], it doesn't differentiate between modeling and implementation. The real challenge of "implementing" this project was a conceptual one. We needed to translate the requirements into a domain model.

In this report, we illustrate how modeling, implementation and interface

design effect each other and how they evolved through this interdependence. Because the interface maps the underlying model, we often illustrate the functional structure of different interfaces through simple interface schemas as opposed to arbitrary concrete mockups or screenshots. The subsequent master thesis will involve more subtle variants and screenshots.

1.1 Goal

In the preceding master seminar [3], we compiled a list of 36 requirements for the interaction design of music composition interfaces. We partitioned them into 4 broad categories:

1. Simplicity
2. Freedom
3. Exploration
4. Abstraction

The abstraction category turned out to be the largest and most domain specific one. It is central to music composition interfaces and, therefore, became the focus of this master project.

Our guiding set of requirements is rather comprehensive. For this project, we needed to further prioritize, even within the abstraction category.

To tackle the requirements that have the most promising "Return of Investment" we selected ones that *a*) address the core of the interaction model, *b*) have vast implications for user experience and *c*) are fundamental for future solutions to other requirement-problems:

Requirement #21 Voice Groups

Requirement #25 Temporal Abstraction

Requirement #32 Preparation

Requirement #33 Global Reuse

These 4 are at the heart of musical abstraction. Requirements #21 and #25 demand that the user can build hierarchies of time intervals and voices. Requirements #32 and #33 then demand an environment that enables the user to actually work with- and benefit from his custom abstractions.

1.2 Terminology

Finally let's clear up some terminology since all the abstraction also merges (simplifies) some terms.

A temporal abstraction relates to an interval on the time-axis [3]. In other contexts we may call it *section* or *part*.

A voice is an abstraction on the frequency-axis [3]. In other contexts, we may call it sound or *instrument*.

Requirements #21 and #25 abolish the principal difference between atomic- and composed entities. This is part of the abstraction. Therefore, we use the term *voice* for atomic *and* composed voices. And we use the term *part* for atomic and composed parts. Whenever we mean something more specific, we specify it.

Also we need to specify the visual language that we employ in object and type diagrams:

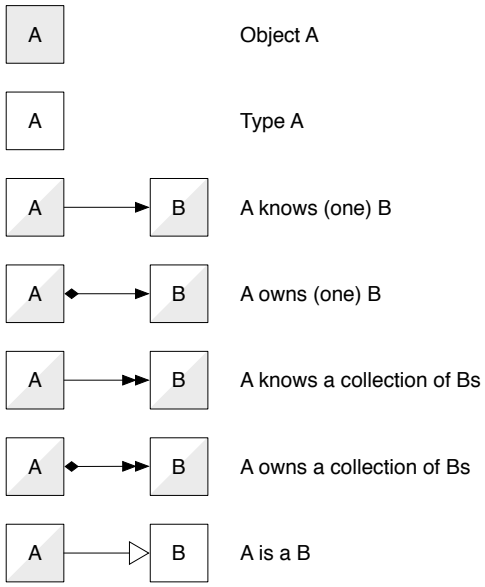


Figure 2: The simple visual language that we used for modeling is **not** UML

1.3 Platform

The platform decision is a design decision. It must follow from requirements.

We carefully considered the platform categories that are most common among end users: desktop, tablet and smart phone. The smart phone was quickly ruled out because, in our context, it has no significant advantage over

tablets that would make up for its smaller screen. So this became a decision between desktop and tablet.

In the end, we chose to design for the tablet because it far better matches the usage contexts as well as user needs and requirements, especially concerning manual input, modalities, mobility, focus and simplicity [3]. In this report, we're already beginning to see how those advantages play out. In the following master thesis, we'll evaluate it more deeply.

2 Development Process

2.1 Piano Roll and Arranger

At the core of common sequencer interfaces are the perspectives of score editing and arranging. We'll briefly explain them adapting the terminology of such software.

Score editing is the process in which the composer edits the notes of one instrument, creating musical patterns called regions. The quasi standard interface for score editing is the *piano roll editor*, which displays notes in the 2-dimensional space spanned by key- and time-dimension (Figure 3).

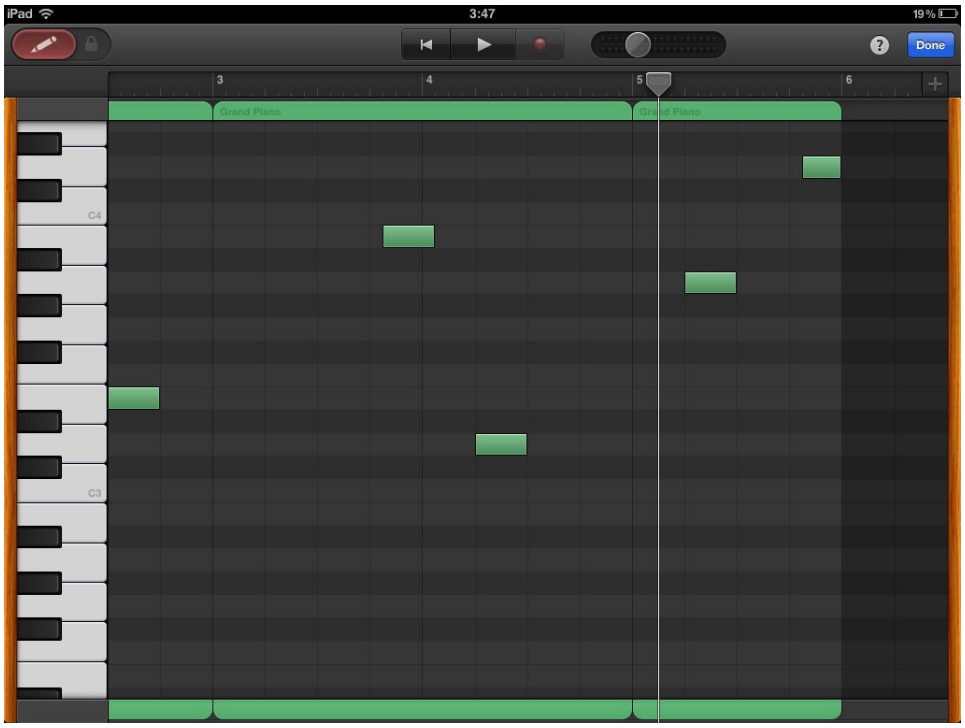


Figure 3: The piano roll editor in GarageBand

We care about the functional aspects of this interface as Figure (4) depicts them.

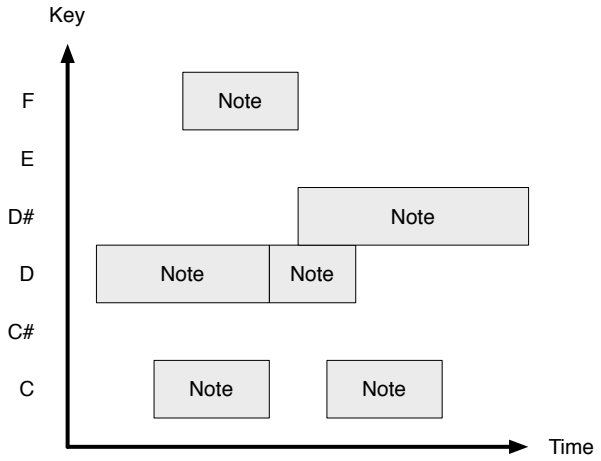


Figure 4: A region is an arrangement of notes in time

Arranging is a similar process on a higher abstraction level. Here, the composer arranges the regions of several instruments, determining the structure of the whole song (Figure 5).

Figure (6) displays the corresponding interface schema. Note that a song can associate several different regions with one instrument. So, regions are different from notes in that they really contain content and not just refer to it. One and the same position on the vertical axis of the arranger view (Figure 6) can actually mean different sounds.

In Figure (4), the two notes on key D produce the same sound because they refer to an audio sample of an instrument playing that key. In Figure (6), however, regions 3 and 4 contain different score data and, thus, produce different sounds.

Figure 5: The arranger in GarageBand

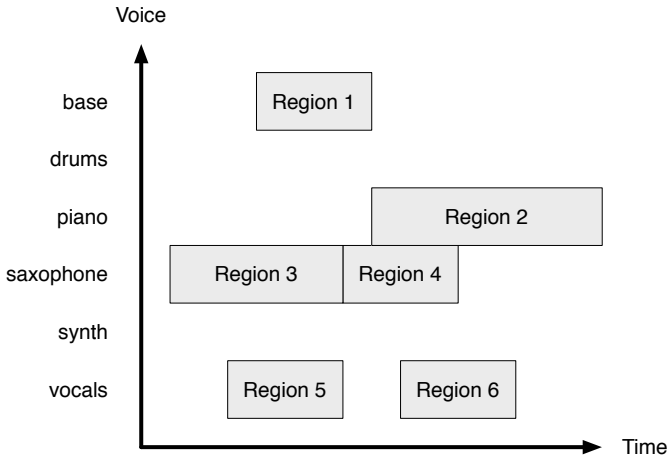


Figure 6: A song is an arrangement of regions in time

Figure (7) shows a sketch of the corresponding domain model. We omitted the details of how a song associates regions with instruments and how regions associate their notes with keys.

This is, of course, a simplification. Unfortunately, traditional sequencers are no that simple.

The model incorporates the notion of an instrument as a sound source for different keys, meaning that a note is a representation of- and reference

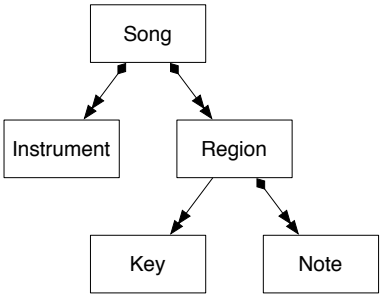


Figure 7: The traditional domain model might look something like this

to an actual piece of sound data, which could be a WAV file.

If the composer wants to use his own raw sound data, he can create *audio regions* that would sit on an *audio track* similar to an *instrument track*. In Figure (6), the *"vocals" track* does not well align with the notion of an instrument. And, indeed, it is more likely an audio track with two different audio regions.

In order to reflect how the traditional approach relates to actual audio data, we must add some polymorphism to our reconstruction of its domain model, like depicted in Figure (8):

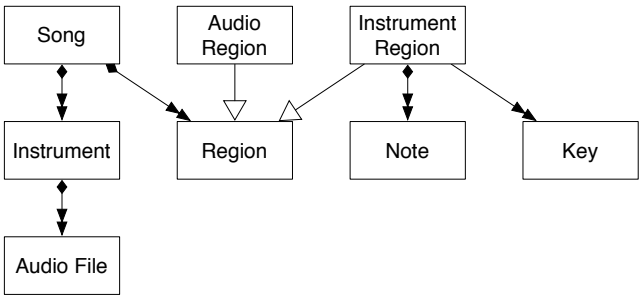


Figure 8: The traditional domain model might actually look more like this

For simplicity, we assume that each audio region possesses its own audio data. In reality, traditional sequencers additionally distinguish between original regions and *region references*. This is were these sequencers really mess things up, and it is impossible for them to satisfy the quite basic requirement #35 [3].

2.2 Integrating Piano Roll und Arranger

We learned how traditional sequencers require the user to distinguish between notes and regions, between score- and audio regions as well as between original regions and region references. We can clearly sense how artificial distinctions lead to an unnecessarily complex domain model which leads to an unnecessarily complex user interface.

For domain modeling, we rather need to derive global patterns from structural similarities than to get caught up in subtle differences that make us produce a soup of clever details.

The basic issue here is that arranger and score editor represent two arbitrary abstraction levels (modalities, scopes, perspectives). Our first step towards our goal of satisfying requirements #21 and #25 was to examine whether these two concepts can be integrated into one. Instead of two separate models for two pre-defined levels we aimed at one model for n user-defined levels. This would simplify the interface and, at the same time, empower the user.

There are two basic analogies that we utilized for the integration:

1. Instrument regions contain score data which is an arrangement of notes. The arrangement of regions within a song is, conceptually, score data in its own right, traditional sequencers just don't treat it as such.

   The other side of that coin is that notes are practically audio-regions in their own right since they reference pieces of audio. Of course, traditional approaches neglect this as well.

2. Arranging and score editing are editing activities in the frequency-time plane. For the time dimension, this is obvious but what about instrumentation vs. keys? Both are distinctions within the frequency dimension as we explained in [3]. There, we also discussed how every entity of tonal music can be associated with a base key. Aside from their different scales, regions and notes stem from the same musical concept.

Regions as well as notes are events in time that reference different sounds. If we generalize Figures (2) and (6) in that sense, we end up with an instance of the frequency-time interface like depicted in Figure (9):

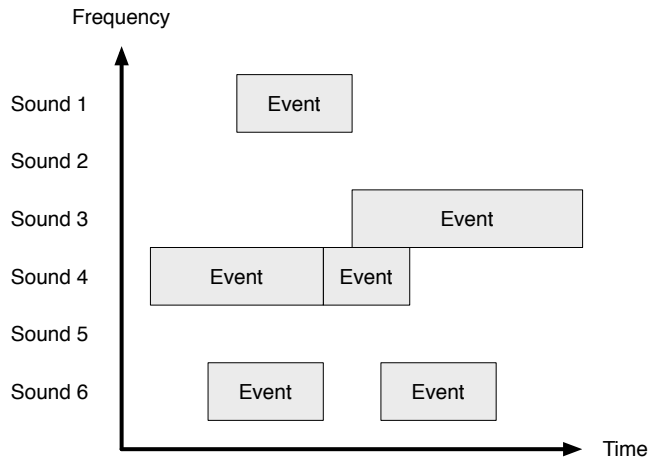


Figure 9: The frequency-time editing plane is a design convention that reflects some fundamental facts about music composition

The corresponding domain model might look something like this:

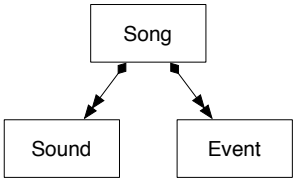


Figure 10: A song associates events with sounds

The one thing that we inherit from conventional sequencers and which will provide some familiarity to the user is this editing in the frequency-time plane. This is the starting point of our design process as well as the user's learning process.

2.3 Structural Recursion

Our model, as shown in in Figures (9, 10) is simpler but actually not as powerful as the traditional one that we want to improve on (Figures 6, 8). The user would now only have one abstraction level to work with where, before, he had two.

So, how would we recreate regions containing notes using our model? A sound representing a region would have to contain an arrangement of sounds

representing notes. There can be any number of recursion levels in an object graph, as Figure (11) alludes:

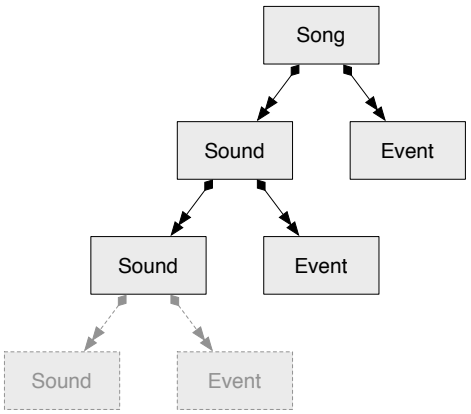


Figure 11: Sounds may contain other sounds.

Because a sound is actually composed of other sounds, we call it a *composition*. Figure (12) shows the resulting class diagram:

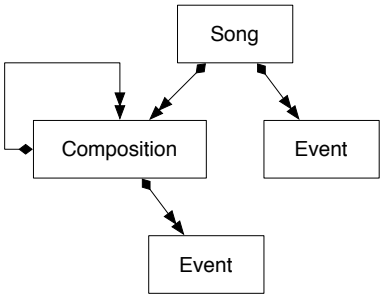


Figure 12: A song associates recursively structured compositions with events.

To be consequent and to satisfy requirements #32 and #33, we acknowledged that the idea of a *song* is just a convention referring to an arbitrary special case of composition. Adhering to the results of our previous work [3], we enabled the user to apply his own understanding of what he is working at. He would model a "song" as a composition:

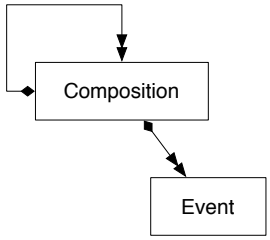


Figure 13: There are only recursively structured compositions.

Now we can deliver both, simplicity and power, from the model through the interface to the user:

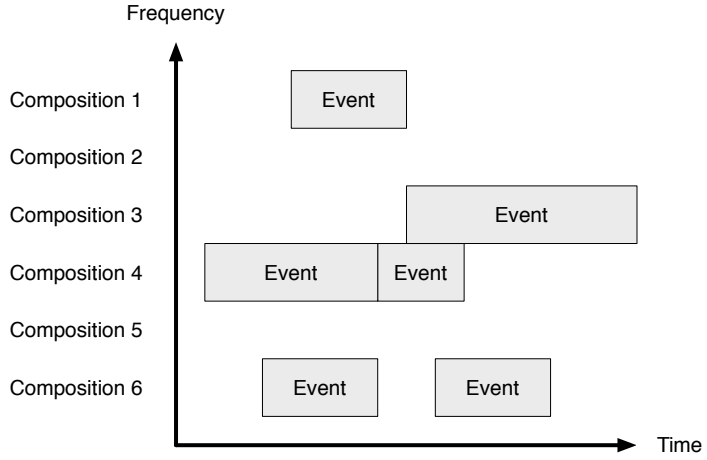


Figure 14: A composition associates events with its sub-compositions

The structural recursion follows directly from how music works [3] and from the requirements we identified. In whatever way we may construct, design and implement it, we need a tree-like model of composed musical structure. The beauty of such a model is that it not only enables us to satisfy the four core requirements but also that it helps us to meet most of the other 32 requirements later on.

The fact that arranging now works like score editing also has the advantage that all material, an all abstraction levels, can be edited through metaphors and gestures of drawing. Previously, this was only possible on the level of notes, i.e. in the piano roll editor. Drawing in the arranger would only create empty regions which the user then would have to "fill" with score data.

Letting users create content directly through drawing adds great value to any CST as we previously elaborated [3], and it was another reason for us to choose tablets as our platform.

For instance, Resnick and his colleagues [4], including Shniderman, demanded as one of 12 design principles for CSTs that we "Design for Designers". We should let users sketch ideas by drawing with their hands to support their "creative reflection" and "reflection-in-action" and we should enable this even in "non-diagramic domains, such as writing and movie-compositions" using "two-dimensional spatial positioning as a representation". How could we ignore this advise regarding the time-frequency plane of music composition?

2.4 Naive Abstraction

Due to recursion and generalization, the model/interface that we arrived at was quite powerful and, yet, simple. But it also had its quirks and we call this first iteration as what it is: *naive abstraction*.

In this section, we uncover the core challenge that was seeded within the requirements and that our design process had to overcome. The smallest example to illustrate this is a piece of music that makes only one distinction in each of both dimensions.

Consider a song that involves two voices (for instance, vocals and harmonies) and two parts (for instance, one verse and one chorus). How would the user model this? Within the song-composition, he could either create two voice-compositions that would contain part-compositions like in Figure (15), or he could create two part-compositions that would contain voice-compositions as in Figure (16).

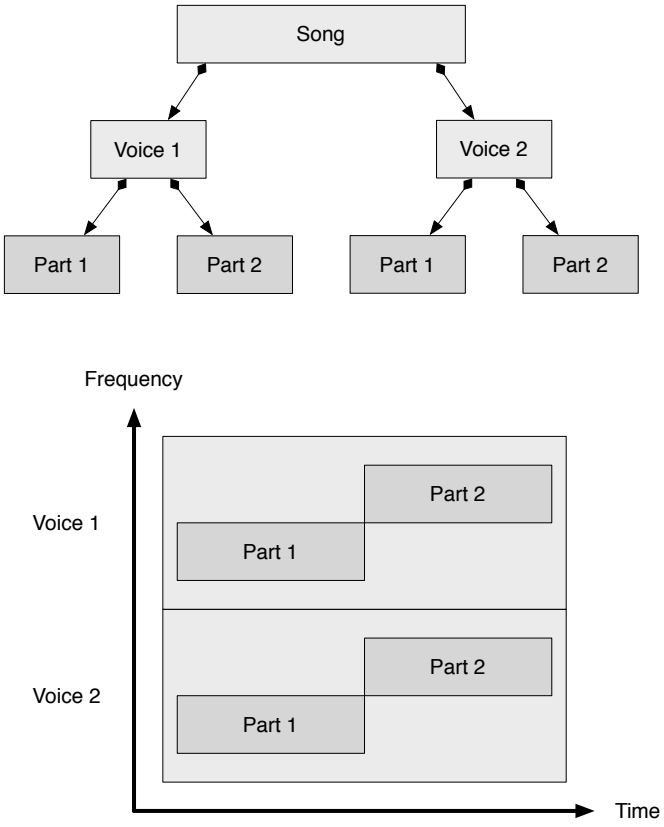


Figure 15: Naive abstraction utilized for working with voices (instruments)

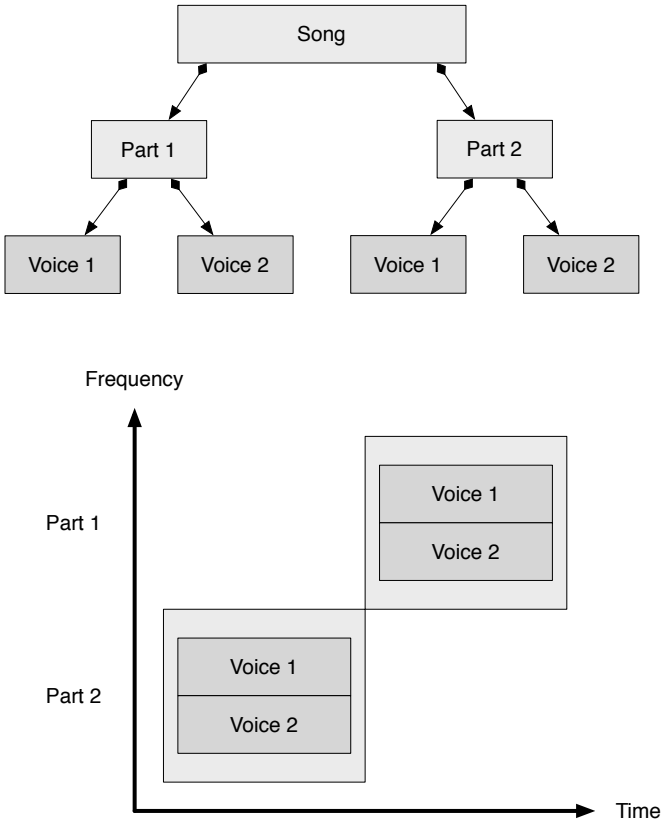


Figure 16: Naive abstraction utilized for working with parts (time intervals)

The figures show the object graphs and their corresponding interface instances. In the illustrations, the nested compositions are pictured in darker grey, but they wouldn't necessarily be visible, depending on how the interface would be designed in detail.

Neither of both structures is true to the nature of the material (domain) and, therefore, naive abstraction creates problems regarding user experience:

1. The user must decide on one of both structures, although he may not care about them since they are not an innate quality of the material. In that sense, the decision is superficial and only complicates interaction.

2. When the composer projects different time intervals onto different compositions like in Figure (16), he somehow misuses the composition-axis. The less repetitive the material is, the fewer events are associated with each part-composition and their super-composition (the whole) reduces to mere concatenation.

   The composer uses the powerful concept of events to encode the simple concept of order, and this conceptual overhead translates to a waste of screen space as Figure (17) demonstrates:

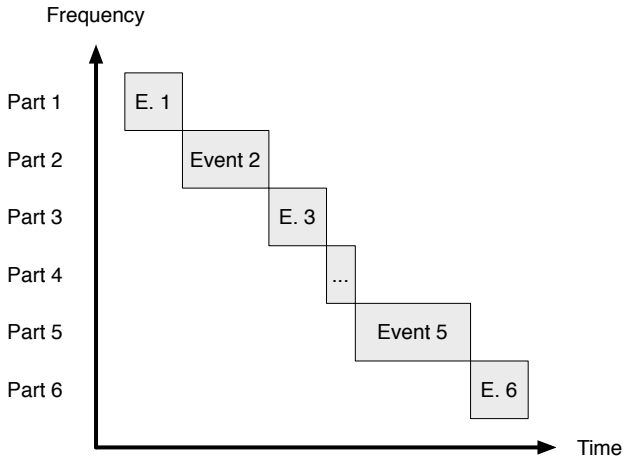


Figure 17: With naive abstraction, concatenation is a mess

3. The issue of wasted screen space is similar to the one of transient voices and requirement #20 [3]. The difference is that, for transient voices, the events always refer to different voices, while the global events in

Figure (16) actually refer to the same voice, namely the composed voice that contains voices 1 and 2.

Even when the voice is atomic, the need to arrange different sub-compositions that refer to the same voice messes with the interaction concept provided by naive abstraction.

4. Due to redundancy, combining instruments and parts increases the user's workload. For example, in Figure (15), the distinction between the parts is double coded – or lost – depending how you look at it. Making part 2 shorter, requires to do that one change in two different places.

5. Parts are ultimately no entities in the model. Even if time is the primary distinction (Figure 16), the user would need to move two events just to switch the order of both parts. To move part 1 behind part 2 should be just that: *one* move.

6. The user cannot set the editing scope to his needs. For instance, in Figure (15), he cannot choose to arrange (and view) only part 1 across all voices because the corresponding composition entity simply doesn't exist. In Figure (16), on the other hand, he cannot choose to arrange only voice 1 across the whole song.

In summary, the kind of abstraction that each node in the hierarchy represents is fixed. After the user made the decision, the node will represent either a temporal- or a voice abstraction for all future editing. This inflexibility disrupts workflow and undermines the whole idea of custom abstractions.

2.5 Elegant Abstraction

Through this linear streamlined project report, the next step may seem only consequential and self-evident, but it actually took a lot of time, contemplation and sketching before a profound shift of our mindset became this major break through.

The problem we encountered in the previous section bears a lot of resemblance to using folders for file organization. Files might be public or private, work-related or personal, images or texts, important or unimportant and so on. No folder hierarchy can ever reflect this but we can model it with tags. Hierarchical tags are actually a generalization of folders as they can imply other tags just like a folder implies its super-folder.

The composer wants to treat the two basic dimensions of composition independently. So a composition should have 2 independent tags (qualities),

one that identifies its voice (instrument) and another one that identifies its part. Those tags would, of course, be hierarchical and the user would create these hierarchies as a byproduct of composing. Instead of one hierarchy from a recursively defined composition-type, we get 2 hierarchies from 2 recursively defined composition-attributes.

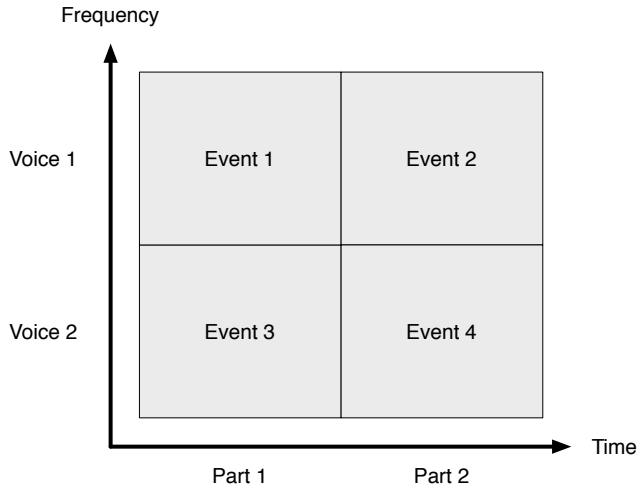


Figure 18: Elegant abstractions makes use of both axes

Now that we identify it by two hierarchical attributes, the composition itself has somehow disappeared. It has become more of a query and less the object of interest itself. The advantage, of course, is that the composer can now work with temporal- and voice abstractions independently from one another selecting any arbitrary combination, and that brings us much closer to our goal of satisfying requirements #21 and #25. The example problems of naive abstraction that we described in the previous section are, thereby, all solved.

At this point, we had a clear picture of how the domain works and how it approximately manifests in the interface, but formally defining the domain model in terms of object- and class diagrams did not work so straight forward. Figure (19) shows a reasonable first iteration of the model:

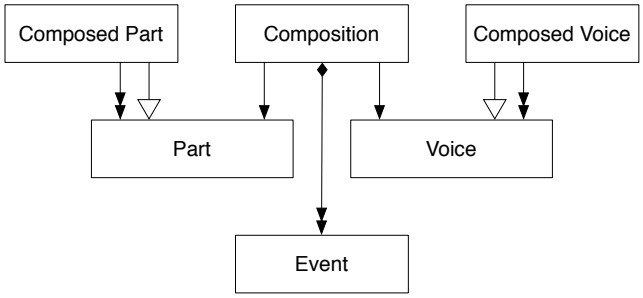


Figure 19: The structural recursion is now in time- and frequency-dimension

A composition associates a part with a voice. Through polymorphism, part and voice might be composed of other parts and voices. A composition also holds events, which make up its actual individual content. An event is a time interval within the part at which the voice actively plays.

The following sections will refine this model by addressing its implications.

2.6 The Library

Note that any part-voice-combination makes a composition, even though it might be empty and not hold any events. To store an empty composition would just mean to bookmark it. Non-empty compositions, on the other hand, need to permanently exist somewhere. Also, since different compositions may refer to the same parts or voices, compositions don't own them, which added the question of where those parts and voices live.

The answer came through the other 2 requirements that we needed to satisfy: #32 *Preparation* and #33 *Global Reuse* [3]. Normally,

> "REPOSITORIES and FACTORIES do not themselves come from the domain" [1].

In our case, however, the library of material is an innate and crucial part of the composition environment, so our model contains types that would otherwise be considered repositories:

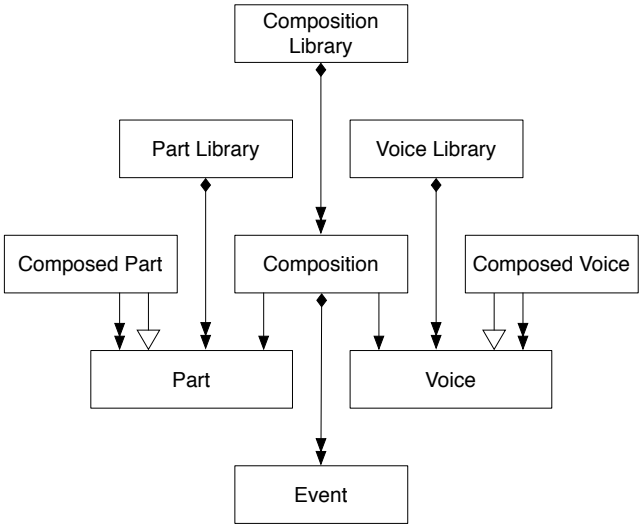


Figure 20: All parts, voices and compositions are at the user's disposal

2.7 The Base Case: Audio Data

We'll keep ignoring the question what events mean on higher levels, where they would refer to composed parts or composed voices. Our model involves recursion and, to make it work, we must first clarify the base case.

The base case corresponds to the leafs in the tree structure of an actual composition instance and it provides the concrete audio data from which all the abstractions (inner nodes) are composed.

So where does the audio come in? In the old way of thinking, composed voices would "forward" events to sub-voices, and atomic voices would, finally, deliver the audio data.

This doesn't apply anymore. Now, the sound or sub-composition that one event indicates depends not only on the voice that the event refers to, but also on the part. While the user can create events by freely drawing in the frequency-time plane, the meaning of an event depends on *both* of its coordinates. In Figure (18), for instance, events 3 and 4 represent different sub-compositions.

Only those compositions whose part and voice are *both* atomic cannot delegate their behavior to sub-compositions. Such "atomic compositions" must link to audio data. To make this association explicit, we made atomic parts and atomic voices explicit types.

Of course, different atomic compositions may employ the same audio data. The repository (library) that holds all audio data performs that mapping: $AtomicPart \times AtomicVoice \rightarrow AudioData$

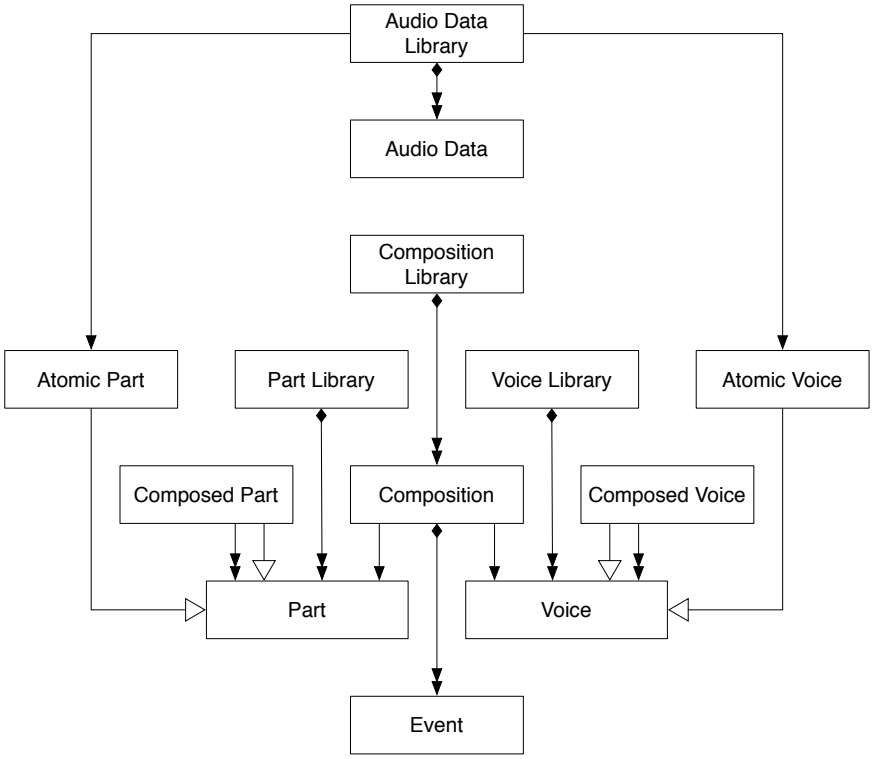


Figure 21: The audio library maps atomic compositions to audio data.

There are many more questions to answer and details to work out, most importantly concerning global events, tonality, editing, reusable scores, navigation and overall workflow. Although we did work on these subjects, covering them in detail turned out to lie beyond the scope of this project.

However, the basic functional aspects are all there. The model we derived allows an application to satisfy the 4 core requirements and more. Interface design alone, without such modeling, could not do this.

This is the perfect moment to return to the user interface. In the following section, we'll develop the interface schema of Figure (18) further to kickstart the evaluation-part of our process.

2.8 Elegant Abstraction in Action

In our example, the user's collection of material is structured as depicted in Figures (22) and (23):

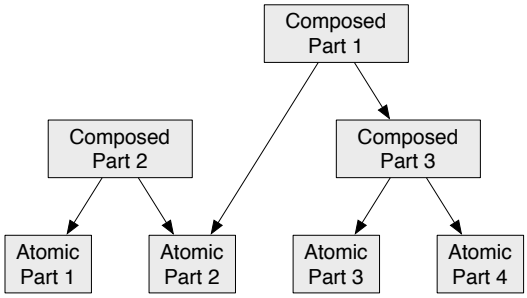


Figure 22: Example structure in the time-dimension

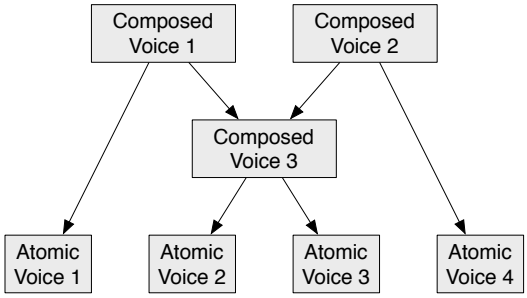


Figure 23: Example structure in the frequency-dimension

The object graphs illustrate two important qualities of the model that hold true for the "trees" in both dimensions.

First, a node can have multiple parent nodes. This means the composer can (re-)use material in multiple contexts.

And second, there can be multiple independent roots. This means the composer can store and access unrelated ideas or projects. "Unrelated" means the nodes might share child nodes (content) but have no mutual ancestor node that would relate them to each other.

One instrumentation or band playing multiple songs would translate to one root voice and multiple root parts. Multiple instrumentations (remixes, versions) of one song would translate to multiple root voices and one root part. In our example, we have two root voices and two root parts.

Of course, each node specifies an order of its children. For parts, this is obvious since they are temporal abstractions, and temporal order is essential to musical structure. Two parts might employ exactly the same sub-parts but put them in different orders.

Sub-voices also need to be ordered, not only for consistency and simplicity of the application, but also for other reasons like interface adjustability, creating composed voices, consistency with traditional sequencers and jux-

taposability (requirement #23 [3]).

What the interface represents depends on what composition (part-voice combination) the user has selected for editing. Figure (24) shows an interface schema for editing the composition made of composed part 1 (CP1) and composed voice 1 (CV1):

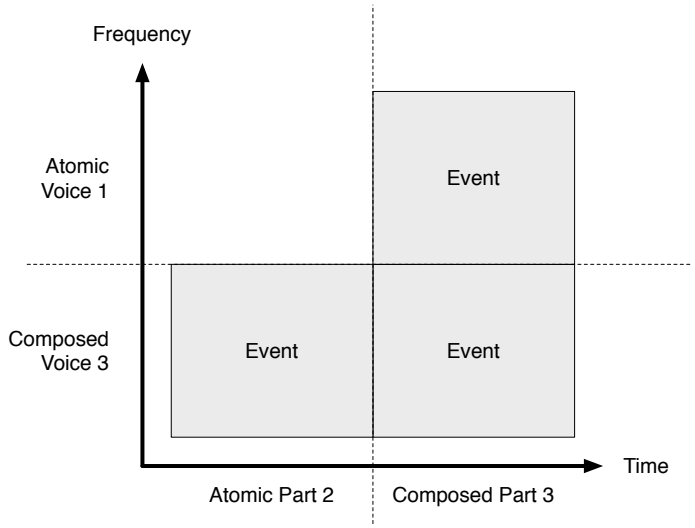


Figure 24: Interface schema for the selection of CV1 and CP1

There is no event for AV1 and AP2, meaning that, in the currently selected composition, AV1 is silent during AP2. The sub-composition that is made of AV1 and AP2 can still contain its own events, but those are not audible here.

The composition relates each event to exactly one sub-voice and exactly one sub-part. The events cannot specify ranges within those sub-entities, the corresponding sub-compositions have to define those details. Neither do events stretch over multiple sub-voices or sub-parts, super-compositions may define such mappings.

This natural quantization promotes high-level editing just as we intend and allows the user to create/delete "composed events" by simply tapping or swiping into such an *event cell* or *quantization cell*.

But we also need an exception to this form of quantization. Suppose that, in Figure (24), the user wants to edit the details of AP2. If he selects AP2, he may end up in an interface corresponding to the schema in Figure (25):

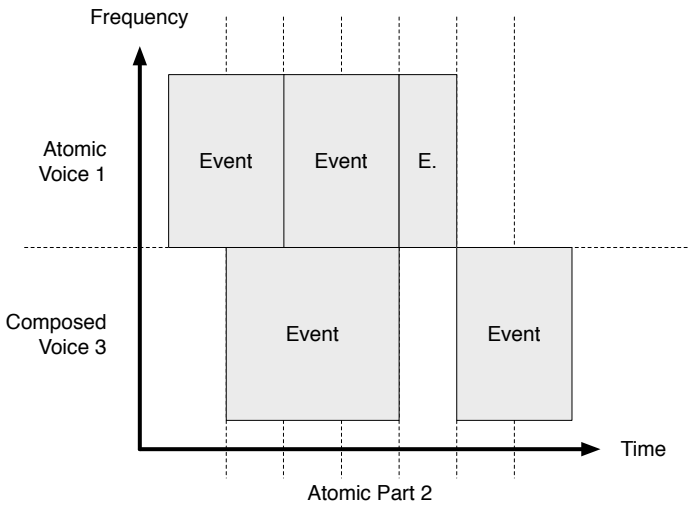


Figure 25: Interface schema for the selection of CV1 and AP2

Now, the currently selected part ist atomic. There are no sub-parts that could be activated or muted for the different sub-voices. Instead, there can be several *atomic events* within the part.

Atomic events directly trigger or represent all the pieces of audio data that are associated with the atomic part and the voice. They do not just mute or activate time intervals like composed events do. Instead, the audio data starts to play from its beginning exactly where the atomic event starts. While the events for AV1 trigger only one piece of audio, the events for CV3 trigger two pieces, one for AV2 and one for AV3.

How editing atomic events is quantized should depend on the zoom level. The minimal visible size of one quantization cell should be big enough for touch interaction. We don't want the user to accidentally introduce micro timing details when editing from a macro perspective. Instead, he can zoom in to comfortably edit on a small scale. The currently applied quantization raster should be visible. In the figures, quantization cells are demarcated by dashed lines.

Of course, the user should be able to freely drag voices and parts around to change their orders. For selection, the interface may provide part- and voice library in additional views that act like file folder trees.

2.9 Simplification

Further insights from implementation and experiments with the interface as well as time constraints led us to adjust and simplify the presented con-

cept. Figure (26) shows a subset of the current implementation as a class dependence diagram:

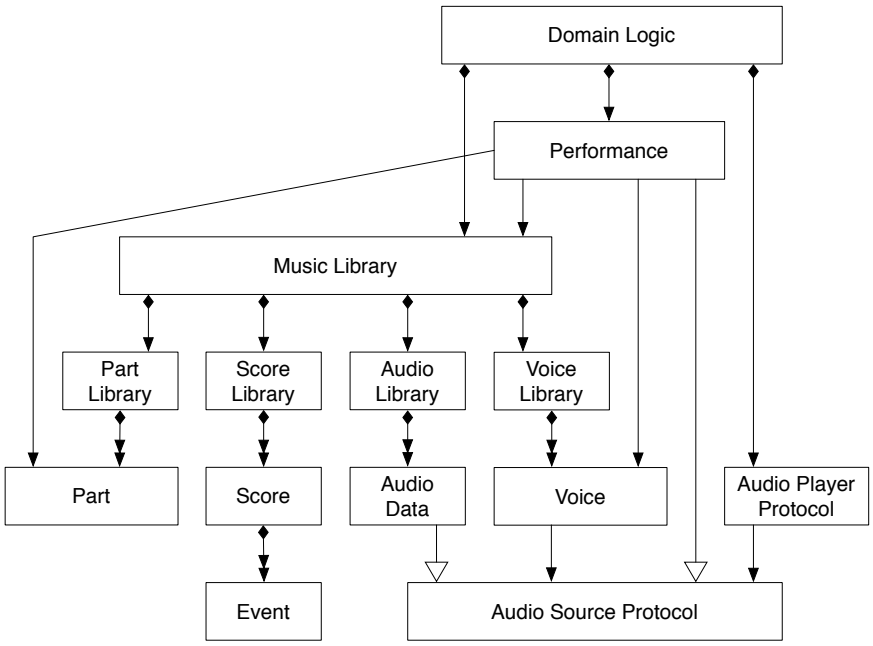


Figure 26: Classes of the domain model and of general audio

These are the basic adjustments we made:

1. We dropped any notion of global (composed) events. This can be added later.

2. We dropped the class-distinction between the atomic and composed variants. Parts and voices are still either atomic or composed, but only as an implicit quality. This is especially necessary when atomic entities are split by the user and, thereby, *become* composed of sub-entities.

3. We dropped the notion of a composition altogether because it is implicit in the model. Instead, the performance is the current part-voice combination, picked by the user from the music library.

4. The audio data library does not know such higher-level concepts as voice and part.

5. All four specific libraries are part of the general music library.

6. The music library maps atomic part-voice combinations to scores. Different combinations may refer to the same score.

Some entities/classes in the diagram deserve further explanation:

1. A piece of audio data is an audio source.
2. For now, scores are atomic and are basically just rhythms. They contain events.
3. An atomic voice knows the audio source which the events of its associated scores would play/trigger.
4. An audio player can play audio from an audio source.
5. The performance can access the music library and acts as an audio source.
6. The domain logic consists of three essential parts: the music library, a performance that uses the music library and an audio player that can play the performance.

2.10 Zooming/Panning vs. Editing

There is no limit to the complexity of one part-voice combination. For example, if the user doesn't create additional voice- and part groups, the performance may consist of many (atomic-) parts and voices, so the editing plane would be segregated into many "part-columns" and "voice-rows". We wanted to solve this base case before designing the navigation in part- and voice-hierarchy.

Such a complex editing plane must allow the user to zoom and pan. Fortunately, the tablet platform greatly supports *fluid* zooming and panning and thereby helps us satisfying Requirement #10 [3].

In Figure (27), the interface has zoomed in on some events of voice 4. Notice how the voice view on the right zoomed only in vertical direction while the part view on the bottom zoomed only horizontally. The panning position of both is also synced with the performance view (top left) that displays the scores (light gray) containing events (white).

In our context, it is important that zooming is "direction-sensitive". In Figure (28), the interface zoomed in only on the parts. Here, the user sees timing information in detail but still oversees all voices without scrolling.

On the other hand, in Figure (29), the user sees voice 4 and its audio samples in detail while overseeing the whole length of the performance, from the beginning of part 0 to the end of part 1.

A challenge that came up here was to keep interaction fluid, modeless, simple and direct while also differentiating between panning/zooming and

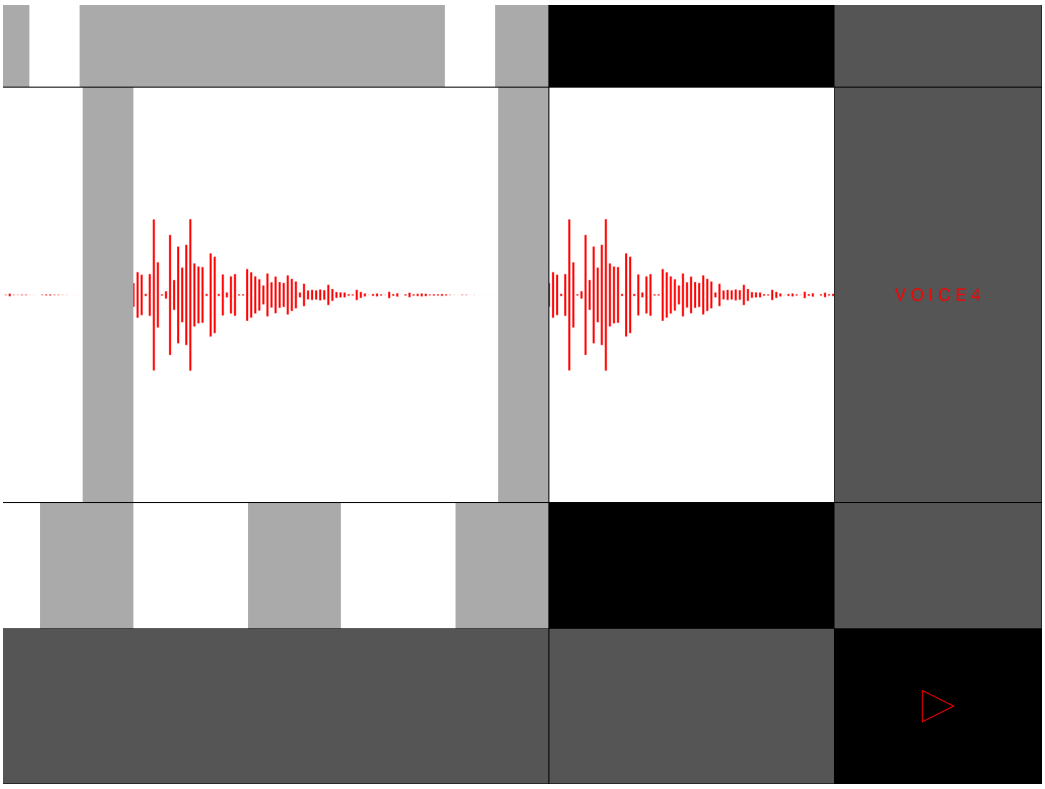


Figure 27: The interface zoomed in

editing. The user should be able to adjust what section of the editing plane he sees and also be able to edit events within the editing plane, all with two fingers. In hindsight, our solution seems natural.

The editing plane resembles a scroll view, and users know how to interact with those. Conventionally, zooming requires a 2-finger pinch, scrolling on the desktop requires a 2-finger swipe and scrolling on the tablet requires a 1-finger swipe.

On the tablet, scrolling requires only one finger because no mouse pointer needs to be moved. This simplification basically reduces the available gestures for editing the scrollable content to 1-finger taps. However, for creative content creation, the user should be able to *draw* content (see also Design Principle 11 in [4]). Furthermore, content creation is the primary purpose, and zooming/panning is secondary.

So, we reserved the single finger swipe gesture for drawing events. Therefor, scrolling (panning) requires two fingers in our prototype. However, once the user "grabbed" the whole editing plane (content view) with two fingers, he can release one finger and still move (pan/scroll) around.

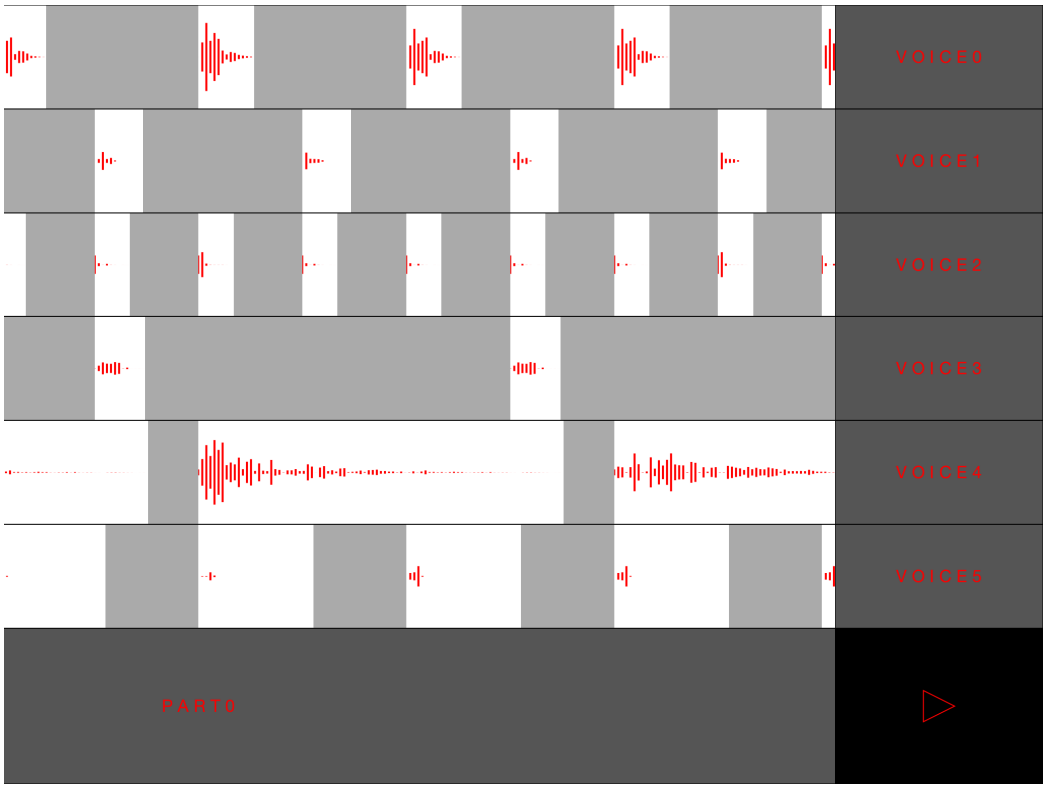


Figure 28: The interface zoomed only horizontally.

Here, inertia scrolling becomes an essential ingredient of the interaction. In our prototype, the user can temporarily release all fingers and still proceed scrolling with just one. Only when all fingers are released *and* the motion of the editing plane has been stopped, does the interaction switch back to editing.

To make this work required some fine tuning. For example, the interface shouldn't zoom while the user scrolls with two fingers. And it shouldn't accidentally scroll a little when he intents to return to editing by stopping inertia scrolling with a short tap.

There are two situations in which it isn't immediately obvious whether the interface is still in scrolling mode. One is simply the inertia scrolling decelerating. Requiring a minimal scroll speed on touch down to proceed panning would restrict 1-finger scrolling and, therefore, only utilize a trade-off.

The other situation occurs when the viewport hits the edge of the editing plane and suddenly stops scrolling. In this case, the user might accidentally draw because scrolling stopped so abruptly.

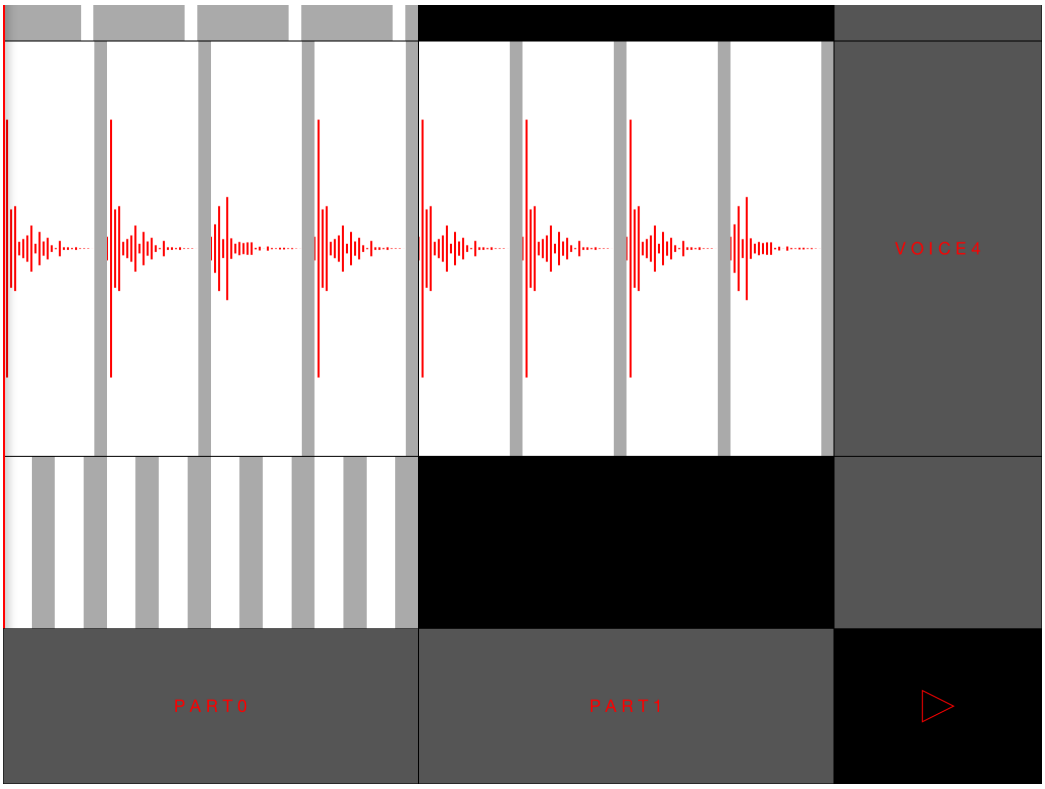


Figure 29: The interface zoomed only vertically.

A possible solution for both situations would be to draw a half-transparent overlay over the whole editing plane during zooming/panning. This would signify to the user that he has grabbed- and is currently effecting the whole plane and is not editing its details. This would go well together with the range overlay of event editing as we'll see in the following section.

2.11 Drawing Events

The editing itself is straight forward. The user can draw events of arbitrary length into an existing score by a simple 1-finger swipe over the score.

The event is not drawn in real time under the finger because the actual effect of the gesture is context-dependent. What the user actually draws is an editing range that marks a certain time interval within the score with a red half-transparent overlay. When the finger is released, one of two things happens:

1. If any events overlap with the range, these events are all deleted.

2. Else (if no event overlaps with the range), a new event is created that matches the range.

This has four advantages:

1. The swipe gesture is multi-functional, and its editing range gives clear immediate feedback on what is being effected in which way. Still, drawing is as simple and intuitive as it can be.

2. The user can cancel the gesture by releasing the finger outside the score in which he touched down.

3. Because the range is quantized, a touch down already creates a range of a minimal meaningful length. This means events can be created or deleted with a single tap.

4. Multiple events can be deleted with one gesture.

In Figure (30), the editing range is being drawn over three events of voice 1. When the user releases his finger, the three events in the editing range are deleted. If the same gesture is repeated, i.e. if the user draws the same editing range again, a new event will be created that matches the editing range, as can be seen in Figure (31).

In the current implementation, the quantization raster doesn't yet scale with the zoom level. When this will be implemented, editing will be even more powerful. The user will naturally adjust the scope of the effect that his gestures have by zooming in and out. He will also be sure that the interactive elements (ranges) will have a minimal size on the screen, appropriate for finger touches.

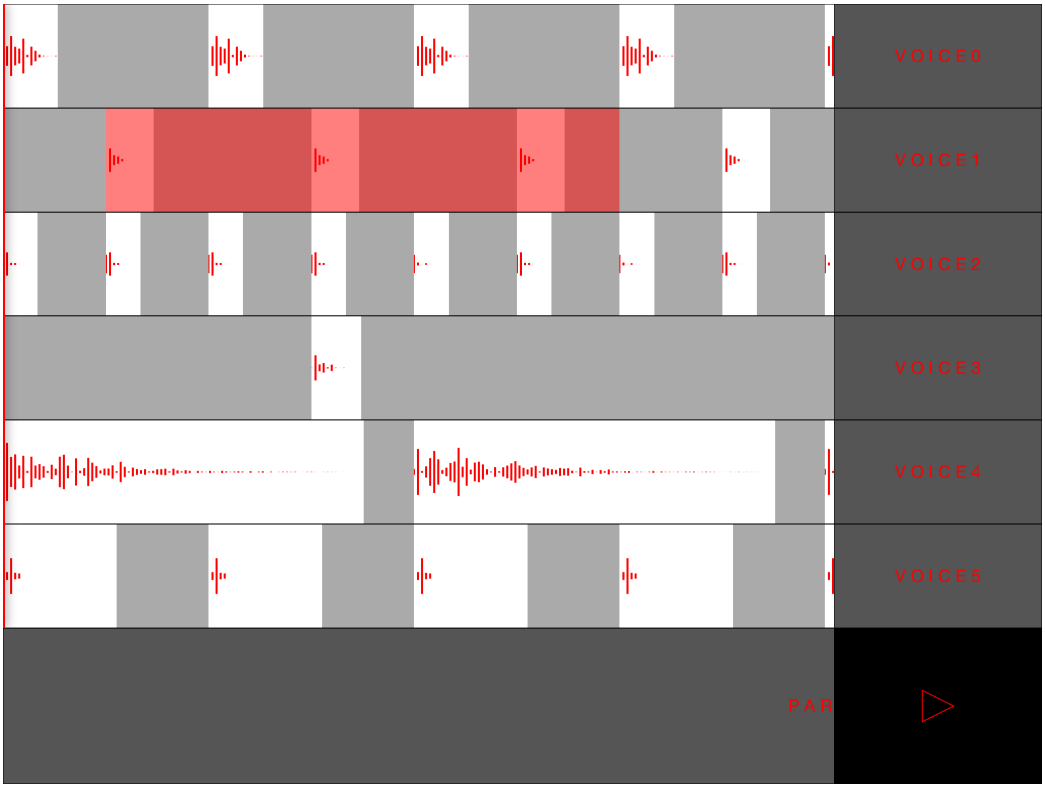


Figure 30: In this case, the editing range will delete three events.

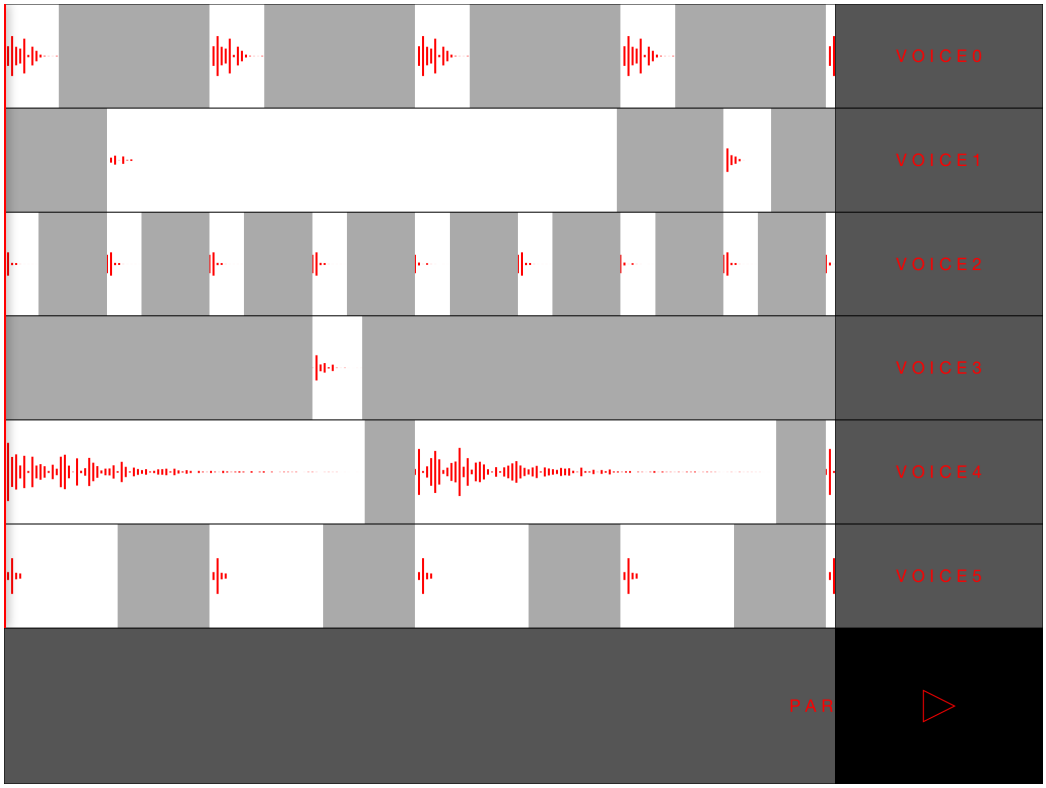


Figure 31: The user created a long event for voice 1.

3 Technical Aspects

Now, we'll discuss some technical issues of our implementation, including those that result from system specifics, i.e. from the platform and frameworks on which the prototype runs.

3.1 System Specifics

We used XCode and Objective-c to implement the prototype for the iPad. The basic audio framework that the code has to rely on is Apple's Core Audio. Fortunately, Michael Tyson published The Amazing Audio Engine (TAAE), a more convenient open source framework that lies on top of Core Audio. We employed it in our code.

TAAE inherits the pull model of audio processing from Core Audio, which means that the destination of an audio stream is responsible for requesting the data it needs from the source. Destinations depend on sources.

To deliver the requested data, a source might itself pull data from yet another source. In this way, several "audio units" can be connected. One unit can use several sources (inputs) to create its output, but it typically serves only one other unit as a source.

So this pull hierarchy constitutes a tree. Its leafs are inputs like audio files, microphones or algorithmic sound generators. The root is typically the device's main audio output.

There are several ways to implement our model and it is, indeed, tempting to directly map the hierarchies in the model onto some pull hierarchies in the audio programming frameworks. However, we ended up dropping this attempt because it intertwined technical details with the domain model. It was cumbersome to implement and made it hard to change the model which is unacceptable with our experimental iterative process.

So we completely decoupled the technical requirements of audio programming frameworks from the domain-driven requirements of the model. To the frameworks, the whole domain model looks like one audio unit. The downside of this approach is that we might miss out on some performance benefits that Core Audio could contribute. However, the connection between model and framework may still be tightened in the future, when the domain model has stabilized.

3.2 Code Structure

The experimental nature of this project demanded flexibility. The code must be easily changeable, even its large-scale structure. Therefor, it must be

clean code embedded in a clean architecture.

The class dependence diagram of Figure (26) depicts a subset of the implementation. This subset only describes the domain model and its dependence on general audio. It is quite independent of system specifics (iOS, UIKit, file in-/output ...) and independent of the business logic and presentation of our specific application.

We explicitly distinguish both dimensions: system dependence and application dependence. Code is more or less system specific and more ore less application specific. These two dimension are independent. So we employ a layered architecture but unlike other layered architectures (for instance the one of DDD [1]) it isn't one-dimensional. Instead, it unfolds in a 2-dimensional space:

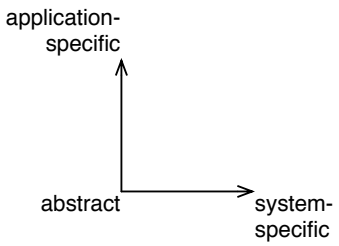


Figure 32: The graphical direction of class dependency arrows in this space (diagram) would convey a specific meaning.

Of course, specific code depends on abstract code. So, semantically, dependency arrows in the space depicted by Figure (32) would only point to the left or downwards. Because a graphical diagram should also be readable, some arrows in a concrete layout might have a drift to the side, in addition to their principal direction (see how the performance uses a part in Figure 26).

Figure (33) shows the Model-View-Controller pattern laid out in our dependence space.

Figure (34) shows what conventional one-dimensional layered architectures do in terms of application- and system dependence. Such architectures have no place for system specific code that is independent of the application. But anyway, they have a hard time differentiating within either dimension.

In the system dimension, we only distinguished model- and system-adaption-layer. In the application dimension, we distinguished portfolio-, domain-, business- and presentation layer as depicted in Figure (35).

To discuss the problems of traditional architectures, the benefits of our dependence space and the meaning of the layers that we picked would be a paper in itself. So for now, we just point out that, in the architecture

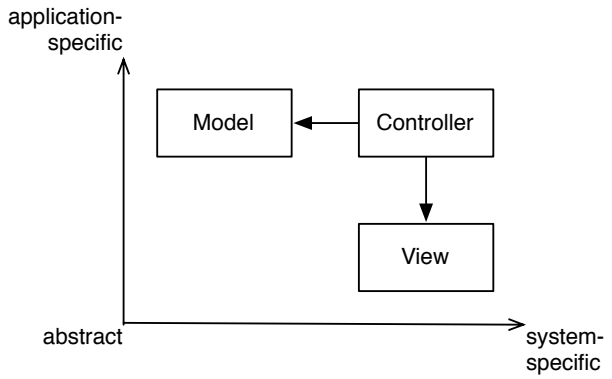


Figure 33: The distinction between application- and system dimension puts the MVC pattern and its dogmas into a much clearer perspective.

of DDD [1], the domain model layer sits on top of the infrastructure layer. Thereby, the DDD architecture profoundly violates the Dependency Inversion Principle and subverts a defining insight of DDD, which is that the model should be isolated and independent.

In the following, we'll illustrate our code through additional class dependence diagrams. One diagram showing all classes would not be readable, so we go through the different subsets that were identified by Figure (35).

Figure (26) showed the domain model together with the model portfolio. The latter comprises a simple audio data class as well as the interfaces (protocols) for audio players and audio sources. The interfaces enable system specific code to serve model objects. Thereby, model objects can trigger the use of system specific services, while model classes are still decoupled from the system.

The implementation of the audio player protocol as well as an audio file class belong to the system portfolio and use a wrapper class of TAAE to access the system's audio capabilities. The whole portfolio layer is depicted in Figure (36).

We already presented the domain *model*. The domain *system* only contains a domain controller that is responsible for 1.) getting audio data from files into the domain model and 2.) injecting the audio player implementation (TAAE Audio Player) into the domain model (dependency injection).

A domain is an area of application and not the application itself. Therefore, it is the business layer that provides the application specific use cases. In our implementation, this layer is still very thin. The only significant use case of the prototype is the creation of musical content for testing.

Finally, the most application specific layer is the presentation layer. It is

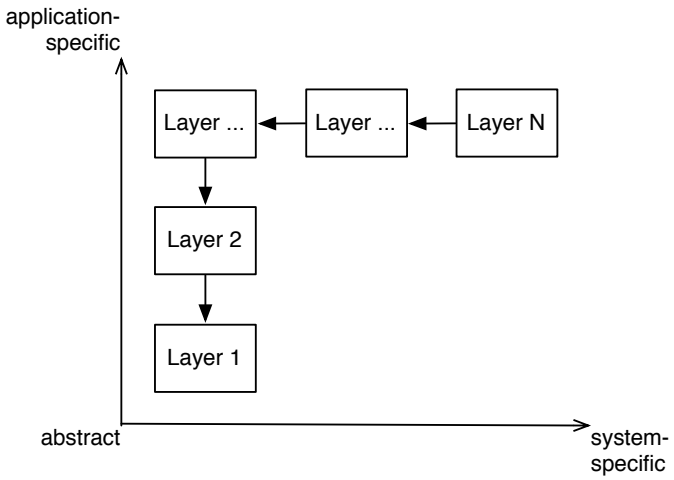


Figure 34: A conventional architecture of N layers

laid out in Figure (37). This layer presents the domain and its use cases to the user. It also transmits user input back to business- and domain layer.

The presentation logic handles those tasks independently of system specifics. It could just as well be implemented in standard C++ without additional libraries. It defines what the user sees and how the screen is laid out. It also defines the user interactions to which screen elements can respond and translates that input into use cases or domain operations.

The presentation system uses the UIKit framework to implement the presentation logic for iOS.

There is a symmetry in the presentation layer that reflects how general presentations correspond to specific UIView classes. And, although it isn't expressed in the diagrams, there is also a correspondence between presentation logic and domain model. Most presentations are there to present some domain model object.

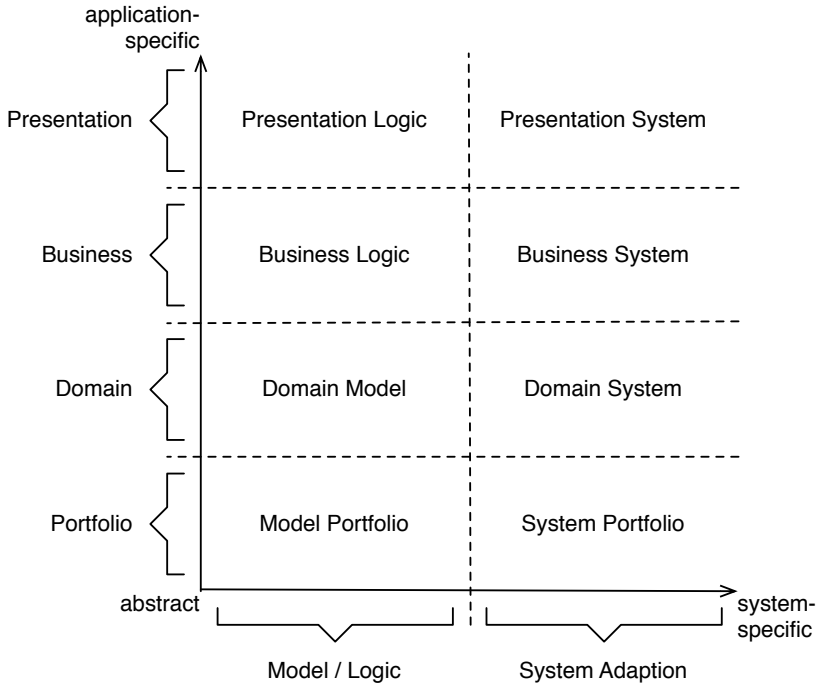


Figure 35: The architectural layers of our implementation

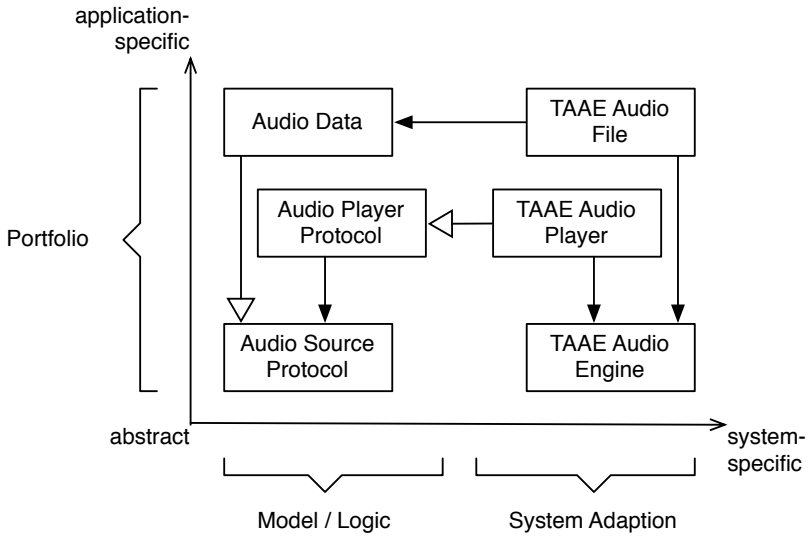


Figure 36: The portfolio layer of our implementation

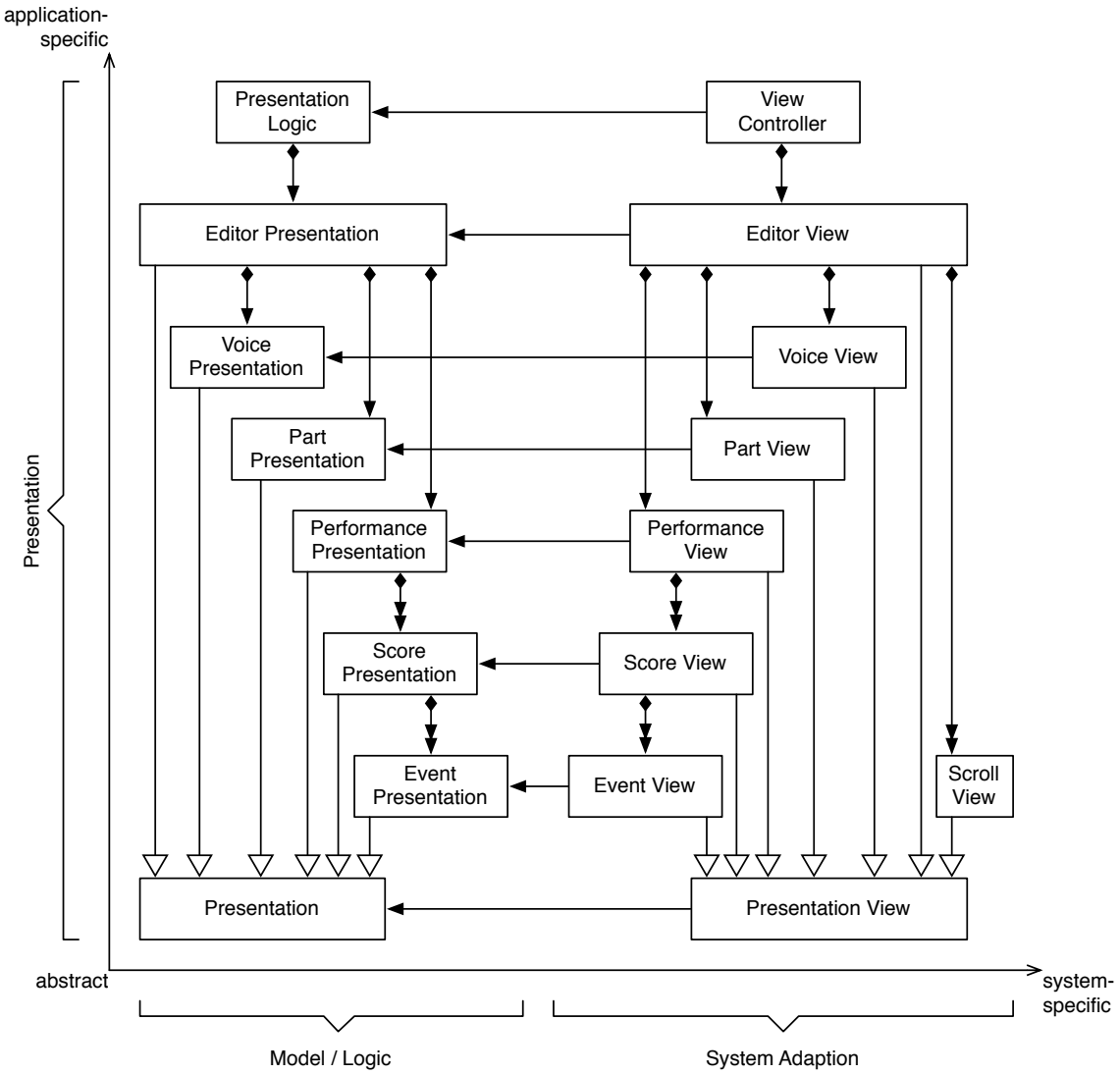


Figure 37: The presentation layer of our implementation

3.3 Hacking and Reinventing the Scroll View

In our first attempt to implement zooming and panning, we utilized the UIScrollView of UIKit. But in the end, Apple's scroll view API didn't suit our needs. These are the biggest issues we encountered:

1. UIScrollView scales its content *after* it has been rendered, so when the user zooms in, the content gets pixelated.

2. The interface should be able to display elements related to whole voices or whole parts at the edges of the editing plane, just like Garageband displays piano keys or information related to instrument tracks on the left. For these views, zooming had to be restricted to one dimension. So their content not only got pixelated by zooming but also distorted.

3. Those marginal scroll views for voices and parts have to be synchronized with the editing plane (performance view). Zoom level and scroll position of all three scroll views need to be in sync. In this regard, the UIScrollView seems to handle both dimensions differently and just wasn't transparent enough to make this work for our application.

4. We could not predict or explain how the UIScrollView interacts with Autolayout, especially in combination with manipulating the content transformation matrix.

5. UIScrollView doesn't let us configure the way it zooms in and out. For example, zooming out should shift the focus to the center of the editing plane, independent of where the user does the pinch gesture. Zooming out locally makes no sense at all. Technically, the viewport collides with the edge of the editing plane, which the UIScrollView answers with awkward corrections ofter the zoom out. And semantically, the user wants to regain context and overview when he zooms out.

6. UIScrollView doesn't let us overwrite what gestures are used for panning and zooming.

7. UIScrollView "hides" certain gestures from its content view.

8. UIScrollView does not zoom vertical- and horizontal dimensions independently. We want to differentiate how much the user actually pinched in each dimension so that he can zoom more into voices than parts and vice versa.

We ended up completely reinventing the scroll view, including collision detection, inertia scrolling, zooming and panning. What our implementation does fundamentally different is how it scales the content. We don't just manipulate the transformation of the rendered content view but actually change the view's size before it gets rendered.

To make this practical, all subviews of the content view are laid out with Autolayout. We also adjusted the scroll view and all its subviews in such a way that gestures can be handled in the appropriate content (sub-)view *and* in the scroll view itself. Our solution performs surprisingly well and solved the enumerated problems.

4 Conclusion

The most urgent issue now is a performance bottleneck caused not so much by the amounts of raw audio but by the naive way it is accessed. It not only restricts playback- but also the graphical presentation of audio.

Both problems are quite transparent. We solved them conceptually but have yet to implement those solutions. In short: Realtime audio playback will be enabled through a caching hierarchy that exploits the voice hierarchy. Fast audio visualization will be enabled through an approximation tree (level of detail) for each audio data object. It might also be helpful to cache an image for each audio data object, since the same audio data can appear in many different events on the screen.

The next step for the user interface is to provide the user with ways to navigate voice- and part hierarchy. This navigation is implemented in the domain model and technically solved but not yet reflected by the interface.

In this project, we delivered the proof of principle that the Human-Audio Interaction Lab as we prepared it in [3], is technically feasible and conceptually promising. We also built a foundation that is solid *and* flexible enough for further intense experimentation.

There are, of course, countless implementation tasks that haven't yet been done at this point. But, fortunately, this is an experiment in progress, and it's supposed to go on for the subsequent master thesis and beyond.

References

[1] Evans. *Domain-Driven Design: Tackling Complexity In the Heart of Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[2] S. Fichtner. Direct manipulation. Seminar: *Theories and Models in HCI*, `http://hailbringer.com/writings/dm_seminar_paper.pdf`, 2013.

[3] S. Fichtner. What music composition interfaces require. Master-Seminar, `http://hailbringer.com/writings/what_music_composition_interfaces_require.pdf`, 2014.

[4] M. Resnick, B. Myers, K. Nakakoji, B. Shneiderman, R. Pausch, T. Selker, and M. Eisenberg. Design principles for tools to support creative thinking. In B. Shneiderman, G. Fischer, M. Czerwinski, B. Myers, and M. Resnick, editors, *NSF Workshop Report on Creativity Support Tools*, pages 25 – 39. National Science Foundation, Washington, DC, 2005.