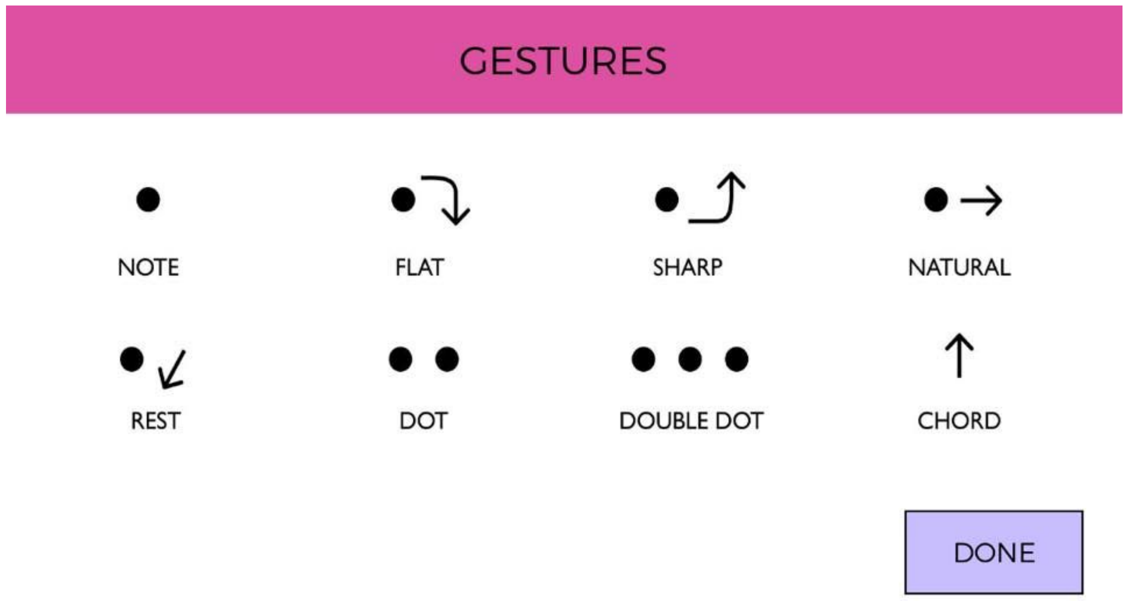
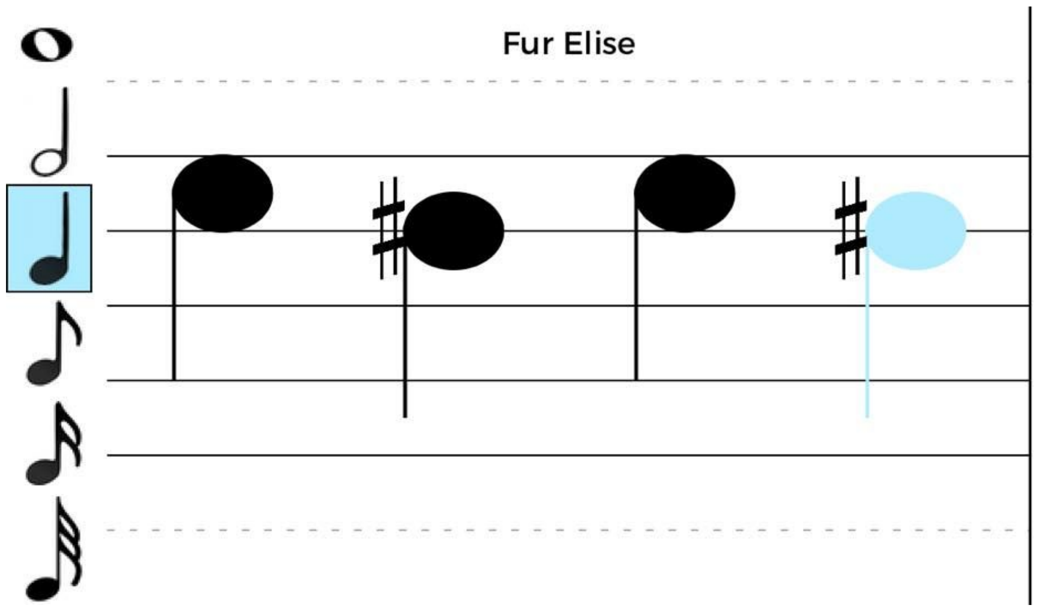
Allegro Composer - Project Proposal

Description of the Product

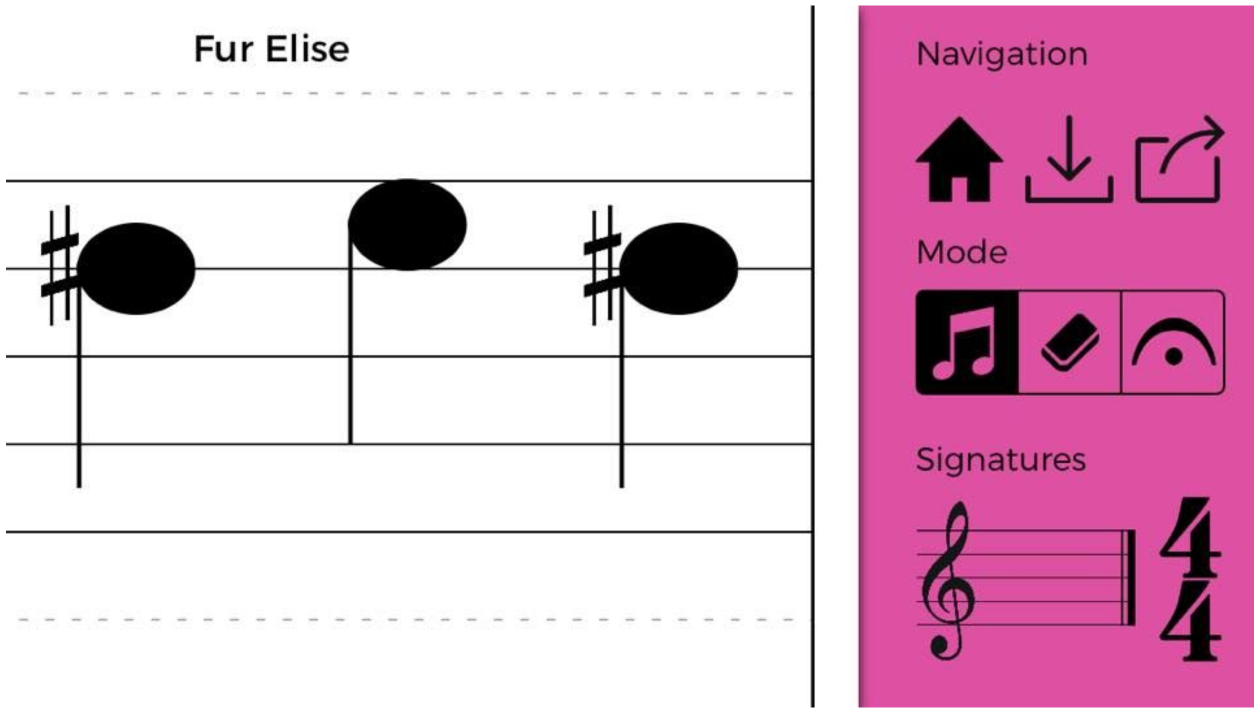
Allegro Composer is a mobile composition app intended for uninhibited jotting of musical ideas. The core feature of Allegro Composer is a set of highly intuitive gestures meant to enhance and quicken the composition experience for users. Allegro is intended to replace carrying around physical staff paper for composers on the go and make composing fast and undistracting.



Allegro features a simple and uncluttered UI to avoid distractions from the idea at hand.



Menus and advanced functionality are shown in pull-out menus to avoid clutter.



Some advanced compositional features, such as articulations, scoring with multiple parts, and certain modifications to notes are not available for simplicity. Allegro is intended for getting down rough ideas that can later be improved upon in other software. For this reason, Allegro features easy export options to pdf or the portable MusicXML format that can be imported into desktop music composition software. Allegro will include basic compositional error checking on accidentals and fitting the correct duration of notes in each measure of music.

Product Need & Competing Products

Allegro is primarily intended for experienced composers and musicians who need to transcribe and save ideas on the go. There are many options for music notation software currently available such as Finale, Sibelius, and MuseScore. These softwares are only available as desktop software or web applications and do not help with mobile composition. Further, they are very feature-heavy and the best use case is in large compositional projects that require lots of time. They do not readily solve the issue of quick jotting of ideas.

Software for mobile devices include Noteflight, Notion, and NotateMe. Similar issues occur in these apps which are mostly designed for tablets and very feature-heavy. All attempt to include many features to make it possible for users to compose complete scores or parts with all possible notations, essentially duplicating the feature sets of heavyweight desktop notation programs. The result is cluttered UI, too many features for small and quick compositions, and serious issues with speed and usability.

Allegro tries to address all of these problems by creating a hyper simple UI and abstracting features into various gestures and simple menus. The result is essentially a “blank slate” program where the user can enter ideas quickly and export and share just as quickly. Because musical ideas can be fleeting, Allegro is designed to be as quick and intuitive as possible so composers can get down their ideas before they are gone. Further, Allegro is optimized for mobile phone screen sizes, solving the “fat fingers” problem faced by many existing composition apps. Allegro’s user interface increases speed and accuracy of note entry by decreasing clutter and increasing screen real estate allocation to music staves.

Because Allegro is designed specifically with mobility in mind, composers will not be forced to carry around staff paper or tablets. Allegro can run on mobile phones which are likely carried around to begin with. Our philosophy is that a 5 second idea should be able to be written down in just as much time. The ease of entry and quick and easy export features will make Allegro an indispensable tool for composers to quickly write down and share ideas. Potential future features may include integration with cloud storage like Dropbox so composers can save compositions direct to their computers and edit with other software later.

Potential Audience

Allegro is designed for experienced composers and professional musicians. Based on very rough estimates using census data and data released by ASCAP and BMI (two of three major publishers representing songwriters in the US), we estimate that there are 150,000 professional composers in the United States and 50,000 professional musicians, bringing the approximate audience size to 200,000 for professionals only. However this discounts the vast number of amateur musicians and composers that may download this app. A quick approximation to the total audience may be to double the number of professionals bringing the total potential user base to 400,000 people.

In the author’s view after 7 years of professional experience playing music in the Bay Area and abroad, musicians tend to be quite behind the curve in terms of technical sophistication. This may be an advantage for Allegro Composer. While other apps include high degrees of sophistication and technical features, Allegro aims to abstract away complications and create the most simple and basic interface possible. It is predicted that the simplicity of use will increase accessibility to digital music notation for the majority of musicians and composers.

High-level Technical Design

How are you going to build your project? Give an overview of the technical architecture of your project. Your description should name the specific programming languages, libraries, frameworks, and other technologies needed for the critical components of your design.

Separation of Concerns

First, we divide the project into two domains:

Graphical Domain

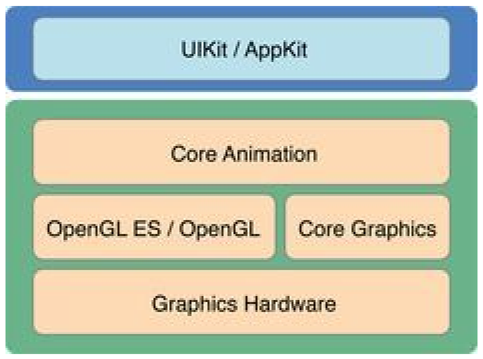
- Governs appearance: We must implement a UI that we display to the user
- Governs behavior: We must recognize gestures and react appropriately

Logical Domain

- Governs fidelity: We must accurately model common western musical notation

Graphics APIs: Libraries and Frameworks

We are implementing the app in iOS. For the graphical domain, there is a wide array of API choices. We could use APIs from UIKit, Core Animation, Core Graphics, OpenGL, or Metal. We have elected to use UIKit, occasionally dipping into Core Animation and Core Graphics. We don't need the performance of Metal. We don't need the cross-platform compatibility of OpenGL.



Our UI will consist of a set of UIGestureRecognizers, organized around an amalgamation of UIScrollViews, UICollectionViews, and custom UIViews and CALayers wherein we will precisely draw vector graphics. When necessary, we will animate using the Core Animation API.

We may utilize open-source libraries to handle many aspects of the UI, but overall the app's core value proposition necessitates bespoke implementation.

Implementation of the Graphical Domain

We will eschew Storyboards and nibs. Since our UI is complex, custom, and dense, we will write our UI in code. Storyboards and nibs are nice for getting started, but they can't be code-reviewed, don't scale well to large numbers of concurrent editors, don't allow for precise layout calculations, and incur performance penalties due increased dependence on Auto Layout.

Implementation of the Logical Domain

We decompose a subset of common western musical notation into the following set of objects:

- Part
    - Contains many measures
    - Holds metadata, like:
        - Composer
        - Comment
        - Title
        - Tempo
- Measure
    - Contains many notes
    - Has a time signature
    - Has a key signature
    - Keeps itself valid by ensuring that notes "fit" upon insertion
- Note
    - Has pitch, octave, duration, accidentals.
    - Can be a Rest

Simplifying assumptions:

- User creates only one musical part. No need for Score object.
- We're not implementing audio, so we can leave out parameters that are required to translate the logical domain into the sound domain.
- For version 1, no articulations, tempo, or dynamics which will simplify UI and modeling

Error Checking in the Logical Domain

We aim to perform error-checking on the durations of notes within measures using rational numbers with the help of a rational number library written in Swift. Right now, we're using our own Swift 3 fork of github.com/griotspeak/Rational.

Export to MusicXML

We will support export of user data to MusicXML, so we will utilize one of the many open-source libraries available to parse and serialize XML on iOS. Candidates:

- github.com/chenyunguiMilook/SwiftyXML.
- github.com/tadija/AEXML

Resource Requirements

What do you need to build your project? Are there any unusual requirements or particular challenges?

Domain Expertise

We need expertise in music theory. Fortunately our product manager is a baller.

Process

We need to budget time for code review. Since two of our members are new to the iOS platform, we will use code review as an opportunity to enable knowledge transfer.

We need to be smart about our branching strategy. One particular challenge is that the team is large and code density is high. By that we mean: a lot of the logic and behavior is contained within a small conceptual “area”. Dividing up the problem into orthogonal/independent parts, separated by well-defined interfaces, will be a challenge. iOS development on a large team can be challenging because of the way that Xcode manages codebase metadata. It’s susceptible to merge conflicts.

Resources

To prevent the build from getting broken, we’re using CircleCI for continuous integration. We need some cash money for this service.

To distribute the app to beta testers, we may use TestFlight. So, we need an Apple Developer account.

Potential Approaches

Are there different ways to deal with the problem area your product addresses? Why did you choose your particular approach?

Allegro was born out of a need for mobile composition. To achieve true mobility, composers should carry as little as possible. Allegro was designed as a mobile phone app because this would eliminate the need for composers to carry tablets or staff paper. In this sense, a mobile phone app is the only option for digital mobile music composition that takes advantage of existing technology that users are likely to be carrying with them already.

Objective-C versus Swift

	Objective-C	Swift
Pros	● Larger set of existing libraries ● Can use C++ ● Ostensibly, we wouldn't have to write our own MusicXML serialization and parsing ● 1 of our 5 members is stronger in Objective-C than in Swift	● Easy to ramp up the two team members who are new to iOS
Cons	● 1 of our 5 members is stronger in Objective-C than in Swift ● Would be dependent on a third-party binary for customization, potentially limiting the customizations we can perform to meet product requirements	● Must write our own logic to serialize and parse MusicXML. (There are, however, XML libs available)

Rendering with Objective-C++

In theory, we could utilize the existing sheet music rendering library: MusicKit (Objective-C++). However, our app demands an unusually high-level of user-interactivity. Interfacing with another library in another language and trying to make parts of the layout interactive could get real messy real fast. We've determined that we're better off writing the layout ourselves and keeping the code simple, compact and readable.

Musical Notation Libraries

We could try use an existing library for handling the logical domain. There’s one in particular that’s a work-in-progress and feature-rich. Instead of writing our own, we could figure out what’s missing from the existing implementation and extend it if necessary.

However, for our use case, we don’t need every single aspect of modern musical notation. Our simplified assumptions make it possible to omit many complicated edge cases. We can write a small library that’s focused on two main tasks: (1) easy and safe mutation in response to UI inputs and (2) reliable export to MusicXML format.

Assessment of Risks

What are the potential risks involved in your project? What can be done to reduce the risks?

Mixed iOS experience

Three of us have iOS experience. Two of us are new. To ameliorate this issue, we've come up with a strategy to onboard the team members who are new to the platform.

Based on the product requirements and our implementation plan, we were able to identify a promising way of partitioning the work. The app can be broken down into two parts: the graphical domain and the logical domain.

The graphical domain is the UI. The logical domain is our system of musical representation.

Success in the graphical domain depends heavily on our understanding of the iOS UIKit API. Success in the logical domain, however, does not depend on experience in iOS. Instead, it depends on our ability to represent music in code. In that way, the logical domain represents a more "pure" software development challenge.

Name	Music Theory Experience	iOS Experience	Graphical Domain	Logical Domain
Brian (TL)		Yes	Yes	Yes
Priya	Yes	Yes	Yes	
Qingping		Yes	Yes	
Nikhil	Yes			Yes
Kevin (PM)	Yes			Yes

To this end, we've split up our team of 5 into two teams of 3. The two individual contributors who are familiar with iOS are focusing on the graphical domain and the two individual contributors who are new to iOS are focusing on the logical domain. Brian, the tech lead, is the 3rd member of both teams, playing a support role in both domains. Kevin, the product manager, is keeping a close eye on the graphical domain to make sure our implementation meets product requirements.

Naiveté

We could be wrong about...	We can reduce the risk by...
The difficulty of serializing and parsing MusicXML	Designing and prototyping this feature upfront
The difficulty of implementing our UI (which features a wide array of interdependent gestures and behaviors)	Scoping out a subset of the features and prototyping. Then, iterating.
The difficulty of accurately rendering notes without sacrificing aesthetic appeal	Trying different rendering techniques to select for the best balance between accuracy and UX
The difficulty of modeling the logical domain	Implementing a subset of the features and getting it working end-to-end with the UI. Then, iterating.

More generally, we can reduce risk by breaking up the project into smaller-scoped, gradually refined versions. Our first version will be a rough draft in which the whole system is made to work end-to-end. The riskiest parts are prototyped in order to better understand known unknowns and identify unknown unknowns. In this first version, the dragons are lured out of their lairs. In the second version, with a better understanding of the challenges and risks, we will refine our UI and implement features which lend themselves to being tacked on.

Next Steps

What do you need to do next in order to build your product?

1. ~~Concretize the desired feature set.~~
2. ~~Decompose the problem into independent parts.~~
3. ~~Break down the software development cycle into an iterative process of refinement.~~
4. Write an initial prototype over a two week period:
    - UI team prototypes the rendering system and implements app navigation.
    - Logic team implements a basic system to represent musical works in code.
    - We connect the two layers together.

Length: ~2000-2500 words