# Chorus: End User Programming of Social Applications

by

Jodie Lian Chen

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Bachelor of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2017

Author . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Department of Electrical Engineering and Computer Science
February 3, 2017

Certified by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Daniel Jackson
Professor
Thesis Supervisor

Accepted by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Dr. Christopher Terman
Chairman, Department Committee on Graduate Theses

# Chorus: End User Programming of Social Applications

by

Jodie Lian Chen

Submitted to the Department of Electrical Engineering and Computer Science
on February 3, 2017, in partial fulfillment of the
requirements for the degree of
Bachelor of Science in Computer Science and Engineering

## Abstract

Chorus is an end-user programming tool for building mobile social applications, much in the spirit of HyperCard[1] but reimagined for modern cloud-based applications on smart phones. Chorus innovates in two dimensions: first, by supporting collaboration via "social datatypes" that define a shared structured document; second, by providing simple user interfaces on smartphones that support interacting with and designing these documents. This thesis reports on the design challenges encountered in building these two user interfaces and the solutions that were explored. The primary contributions of this research are: 1) techniques for visualizing and navigating a complex hierarchical document on a phone, and 2) supporting WYSIWYG design of a complex data artifact within the UI constraints of a phone. Qualitative user testing was used to guide and evaluate the design decisions, indicating both successes and problems requiring further work.

Thesis Supervisor: Daniel Jackson
Title: Professor

[1]Hypercard is a visual programming language that allowed people to build Macintosh applications easily.

# Acknowledgments

I would like to express gratitude to Jonathan Edwards, Professor Daniel Jackson, and Alex Warth for collaborating with me on this project and providing endless advice.

# Contents

# List of Figures

# List of Tables

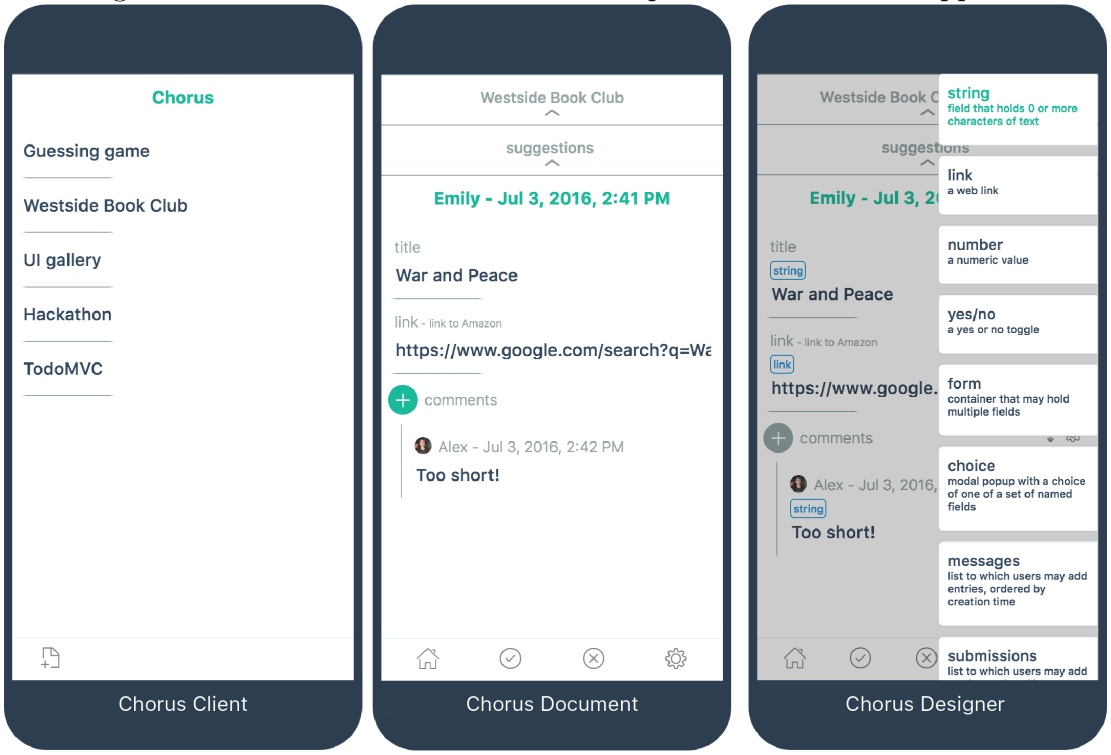
# Chapter 1

# Introduction

Chorus intends to simplify the workflow of planning social gatherings and organizing groups. Planning today's social events requires people to constantly copy and paste information from one application to another. Imagine trying to organize a meetup – the organizer would have to use a subset emails, forms, spreadsheets, chat channels, polls, etc. to get all of the data needed to successfully make this event happen. Chorus reinvents the social planning workflow and the notion of needing a separate application for each step of data gathering. Instead of copy and pasting between many standalone applications, Chorus provides a central "collaborative document," which looks and functions as a mobile application and aggregates the features of the different applications mentioned above. For example, documents may allow users to perform actions such as filling out forms, writing group messages, and voting in polls. By both collecting and storing data in one central location, Chorus streamlines the workflow. Chorus also brings customizability to the end users, the people who organize social events. Using the Chorus designer, end users can create unique, standalone applications that incorporate only the relevant social features. Users can also edit their document and add in new features as they are needed.

We have built a standalone desktop application as a prototype that mimics a mobile application UI. We decided to focus on a mobile UI, because most online social interactions are now mediated by phones. The prototype offers users the ability to work on a collaborative document together and also to create and customize their own

Figure 1-1: Screenshots from the various parts of the Chorus application



Emily - Jul 3, 2016, 2:41 PM

title
War and Peace

link - link to Amazon
https://www.google.com/search?q=Wa

comments

Alex - Jul 3, 2016, 2:42 PM
Too short!

string
field that holds 0 or more characters of text

link
a web link

number
a numeric value

yes/no
a yes or no toggle

form
container that may hold multiple fields

choice
modal popup with a choice of one of a set of named fields

messages
list to which users may add entries, ordered by creation time

submissions
list to which users may add

collaborative document. User tests were performed to assess usability and pinpoint misconceptions about the UI.

## 1.1 The Parts of the Chorus Application

The Chorus user application consists of three parts: the Chorus client, the shared collaborative documents, and the Chorus designer. Potential users either create collaborative documents with the designer for their social group and they can also use the document to communicate with their social group. Screenshots of each can be seen in Figure 1.1.

### 1.1.1 Client

The Chorus client aggregates all the different collaborative documents a user is subscribed to. From the client, users have the ability to access a document they are subscribed to or create a new collaborative document.

### 1.1.2 Collaborative Documents

The collaborative document allows users to interact with each other. Users make edits to the collaborative document and may publish their changes for other users to see or may cancel their changes. Chorus collaborative documents are based on two concepts: a statically typed tree with data stored at the nodes; and social datatypes, which are the nodes of the tree. The tree structure of Chorus documents is explored more in depth in Section 3.1.

#### Social Datatypes

By studying the types of interactions that users perform in social settings, we were able to extract the social interaction patterns performed and distill them into a collection of social datatypes. These social datatypes, also referred to as "components," are "social" as they specialize in implementing common conversational patterns. They can be stringed together to support the functionality of chat messages, comment threads, polls, and more. A list of components may be found in Section 3.1.1.

Social datatypes are able to function because of states such as error, new, todo, etc. These states, described further in detail in Appendix Two, provide notifications, task tracking, and other standard capabilities that facilitate social collaboration and interaction. States can indicate if a change was recently made by a user or if a certain component is waiting for a specific user's input. The states also come into play for the enforcing type matching. If a user's input does not match the datatype in the document's structure, for example a string when a number is expected (i.e. when collecting phone numbers), the components will produce an error state for users and users will also not be able to publish their changes. Published changes to any of the datatypes sends notifications to other users subscribed to the document, helping facilitate conversation. Notifications of changes or required inputs between participants drive a collaborative workflow. Enforced types also help mediate structured conversations.

### 1.1.3 Designer

The Chorus designer allows users to create the collaborative documents. Users decide which components to add into their document. When adding in components, users are essentially adding in new nodes to the tree structure and deciding what type of data the node stores and whether the node is the parent of a subtree. Since the nodes are social datatypes, the patterns of interaction described in the previous section are built into the document. Additionally, users can manipulate both the structure and the display of the document. This corresponds to moving subtrees and their parent to another level in the tree and how many levels of the tree can be displayed on a single page in the document, respectively.

## 1.2 Summary of Contributions

The main research done is the creation of the UI for the Chorus document and designer. Prior to my research, there already existed a backend for collaborative documents. Content is stored in a tree structure and can be accessed and modified through API calls. However, there was no way for our intended user base to interact with the calls. Creating a successful UI for both allowed us test the validity of our ideas and to better assess the needs of potential users and potential shortcomings of our application. We successfully designed a UI for both the collaborative document and designer and implemented a prototype Electron application. The prototype succeeds in meeting the goals we set out – the document UI allows users to edit documents and the designer UI allows users to edit the structure of documents. The following list is a brief summary of the contributions made:

- The complexity and variety of datatypes involved in Chorus documents could easily lead to a heavily menu-driven interface, which is suboptimal on a phone. Much work was put into avoiding menus and encoding all actions into taps on content items or a bottom icon bar.

- How is a complex hierarchical document to be visualized and navigated on a phone? Different ways of visualizing and traversing the document were explored, finally settling upon a statically indented outline where some nodes of the tree are displayed indented on the same page as their container, but other nodes require a page traversal, leaving behind a stack of breadcrumbs at the top of the screen. The static grouping into pages is established by the document designer.

- How is the user informed of changes that may be scattered throughout the document? Further, some of the social datatypes assign tasks to users to supply certain information – how are those users to be informed of their tasks and given a way to keep track of them? We settled on a representation for changes and tasks using marginal colored bars. We extended this mechanism to also indicate tentative (uncommitted) changes and errors with additional bar colors.

- How are the above notifications tracked across a hierarchical document? We adopted two solutions. First, notifications aggregate hierarchically, so containers show whether any of their contents have changed or contain tasks or errors, etc. Second, to obtain a global view we implemented a "minimap" that shows a birds-eye view of the entire document revealing the hierarchical structure and the attached notifications as well as the location of the current page view.

- It can be as important to see the history of a social interaction as its current state. We developed a "timeline" which shows the history of changes by person and allows going back to that time to see the previous state of the document and what exactly that user changed.

- Database and program design is typically done with textual languages. How can Chorus documents be designed on a phone with a small screen and limited support for typing? This was perhaps the central challenge of this research. We first explored the idea of a "meta-document" that encoded the design of a Chorus document using just the standard datatypes available in Chorus documents. The seductive appeal of this approach is not having to add any new datatypes or UI affordances. But testing indicated a crucial flaw: having to see a document from two views, the end-user view and the meta-view, led to disorientation. We attempted to ameliorate this disorientation by synchronizing the two views on the wider screen of a tablet, but this was not completely effective. We ended up adopting a hybrid approach, where designing is done "WYSIWYG" in the context of the end-user view of the document, but by expanding one item at a time in-place to reveal all its designable attributes.

## 1.3 Outline

The order of topics presented in this work are as follows:

Chapter two in this work describe related projects that we looked to for inspiration.

Chapter three provides background for the end user parts of Chorus and contributions made to the collaborative document UI.

Chapter four describes the background of the Chorus designer and the design decisions made regarding the designer UI.

Chapter five details the implementation details of the application.

Chapter six describes results from user testing and changes made to the application in response to the results.

# Chapter 2

# Related Work

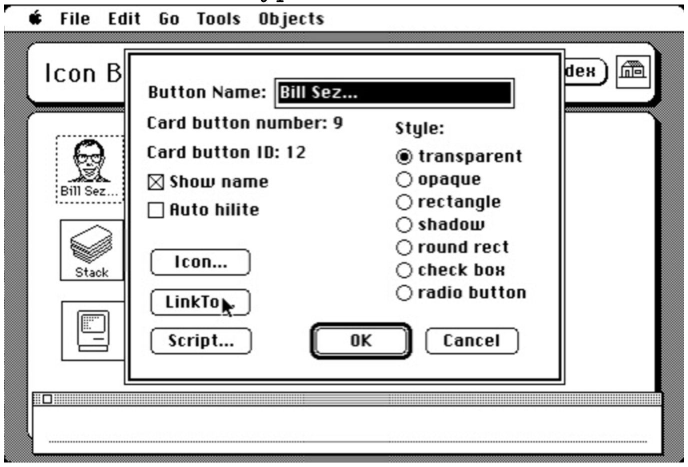
We drew inspiration from many similar projects along our many iterations. As the vision for our project shifted, we found similar, successful applications that we could draw insight from. First we focused on visual programming languages. Later, as we moved away from the simplified programming aspect, we focused on "What You See is What You Get" (WYSIWYG) designer-like applications. This was in an effort to increase learnability of the application, as graphical user interfaces (GUI) are often easier to understand and more familiar to users.

## 2.1 Visual Programming

At first, we intended Chorus to be a means to an end for nonprogrammers to create simple mobile applications with an easy to learn programming language. We wanted the programming to have the same barrier of entry similar to that of spreadsheets or Hypercard, as in people would be able to "program" using the application without actually needing to know any programming.

At its inception, Chorus was meant to be a modern day Hypercard [8]. HyperCard allowed for easy prototyping due to an easy to learn GUI (Figure 2-1) and also came with a simple programming language, HyperTalk, targeted towards beginners. HyperTalk would allow programmers to specify changes to the UI when certain interactions were performed. Inspired by HyperCard, the first iteration of Chorus also

Figure 2-1: An example of the programming GUI from the Hypercard application. Image taken from `http://basalgangster.macgui.com/RetroMacComputing/The_Long_View/Entries/2010/10/23_HyperCard.html`



Button Name: Bill Sez...
Card button number: 9
Card button ID: 12
[✓ Show name]
Auto hilite

Style:
[✓ transparent]
opaque
rectangle
shadow
round rect
check box
radio button

had a programming language that helped specify the structure of UI components for the document. From Hypercard, we also took away the idea of a consistent UI. Each component has its own consistent UI so users do not have to develop the UI, decreasing the barrier of entry in learning how to use Chorus.

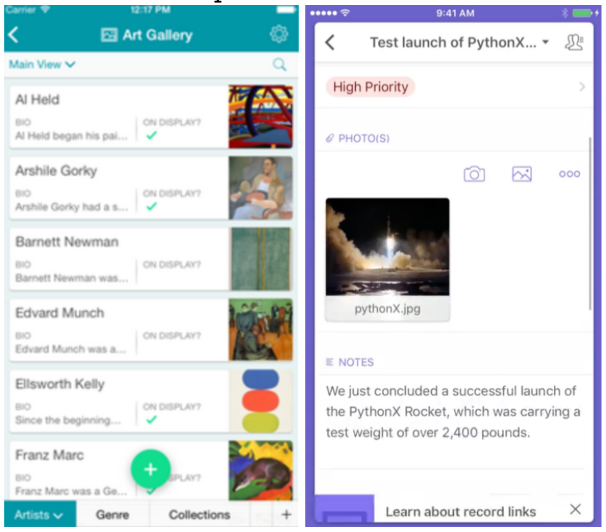
The Chorus code is based on Subtext [2], another visual programming language. Subtext simplifies programming into a form that bears resemblance to spreadsheets. This concept enhances the idea of copying and pasting reused code.

## 2.2 WYSIWYG Designers

After successfully prototyping a programming language based designer, we wanted to assess the usability improvements of a GUI based designer. GUI based editors are easier to learn to use than a new programming language as they offer more user familiarity. From both Airtable [7] and Notion [5], we gleaned ideas for how to integrate the designer options into the collaborative documents.

Airtable, a collaborative database management application, makes it easy to edit data tables. Airtable shares many similarities to Chorus as Chorus is supposed to primarily organize and collect structured data. To edit data table types and content, Airtable's mobile application provides an intuitive UI (Figure 2-2). Users click on the

Figure 2-2: Pictures of Airtable's mobile application. The screen on the right shows a data table in Airtable and the screen on the right shows the screen seen when editing a data table. Image taken from `http://airtable.com`

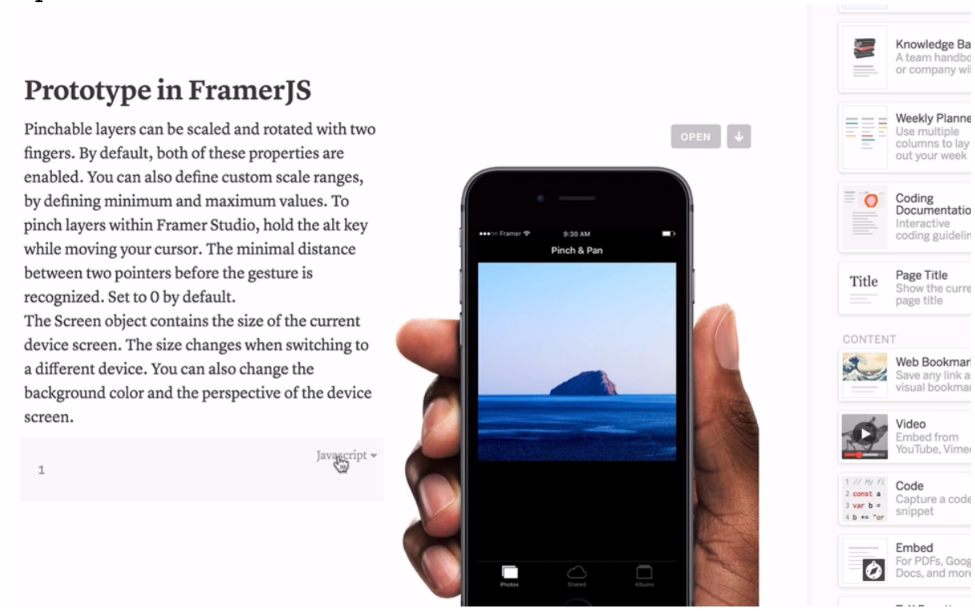


We just concluded a successful launch of the PythonX Rocket, which was carrying a test weight of over 2,400 pounds.

row they want to edit and the application opens a differently styled screen with a card for each editing option. Airtable's editor provided inspiration for how to integrate the designer into the document UI.

Notion also shares many similarities with Chorus. Notion is a collaborative data management and editing platform that offers many of the capabilities of a document and wiki editing platform. Users add in and fill out structured datatypes that offer the functionality of work applications such as documents and wikis (Figure 2-3). Users can drag and drop structured content components, such as videos or code fragments and then fill them out with the appropriate contents. Studying the UI of Notion gave us inspiration for how to transition from the designer into a WYSIWYG editor.

Figure 2-3: Pictures of Notion's document and editor. Users can drag and drop various structural components from the right side into the document. Image taken from `http://notion.so`



## Prototype in FramerJS

Pinchable layers can be scaled and rotated with two fingers. By default, both of these properties are enabled. You can also define custom scale ranges, by defining minimum and maximum values. To pinch layers within Framer Studio, hold the alt key while moving your cursor. The minimal distance between two pointers before the gesture is recognized. Set to 0 by default.
The Screen object contains the size of the current device screen. The size changes when switching to a different device. You can also change the background color and the perspective of the device screen.

# Chapter 3

# Chorus Collaborative Document

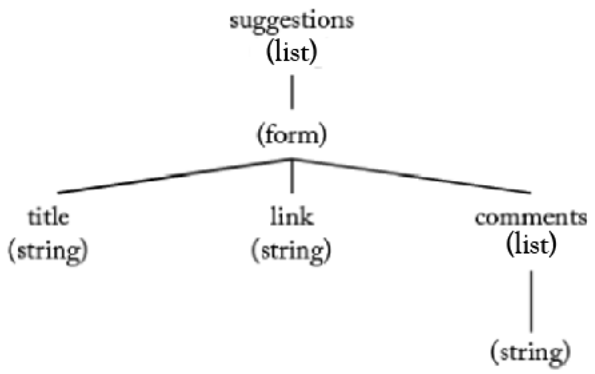
Through our many iterations, we were able to distill down the core requirements we had for the document UI. In designing the UI for the Chorus document, we were challenged with the task of displaying the tree-structured data in an easily digestible manner. We also wanted to make adding to and editing the content of the document intuitive to users. Navigating through the levels of the tree and editing the nodes should feel like going through any mobile application. The next two sections give background on the parts of the Chorus document and the contributions made in designing the UI respectively.

## 3.1 Components

As mentioned previously, the shared collaborative document supports many different functionalities. The various document functionalities we support can be distilled down to the social datatypes defined below. Each component consists of two parts: a label and the components contents. The components below are similar to Lego blocks that can be combined together to create more complex components. This is enabled by the structure of a Chorus document, which is treated as a statically typed tree structure.

The tree structure is similar to a file system tree. Directories can contain files or other directories, and those other directories may contain more files or directories.

Figure 3-1: Document tree

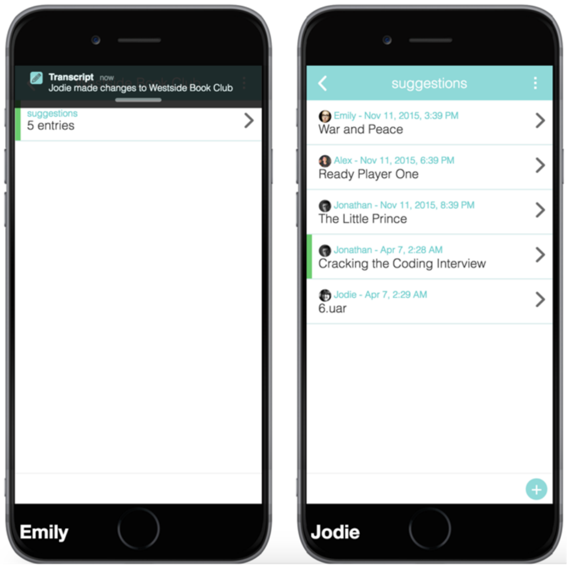


Users explore the contents of the file system tree by traversing up or down the file system tree. Similarly, users explore collaborative documents by traversing up and down the document tree. In the collaborative document we define, this tree structure is created because components can be combined together by certain types of container components (lists and forms, which are defined below). In that case, the latter component becomes the parent of the other in the tree.

An example document used in demonstrations is a book club. Figure 3-2 shows an example of the document UI for the "Westside Book Club" prior to when we started work on the designer UI. A list is a collection of elements of a certain format. In the Westside Book Club document, there is a list of forms called "suggestions." Each form in suggestions has "title," "link," and "comments" fields. Forms are components that consist of fields to fill out. "Title" is a string atom, "link" is a link atom, and "comments" is a list of strings. Atoms are basic units of data. The document tree can be seen in Figure 3-1 and the traversal of the tree can be seen in Figure 3-3. In this document, users submit suggestions that consist of a book title and a link. Users can comment on book suggestions

Figure 3-2 also demonstrates the new notification UI when one user publishes changes to a collaborative document. Notifications are displayed as bars on the left side of each part. In Figure 3-2, the green bar represents a change in a part of the collaborative document. Since "suggestions" is a parent of the atom that was changed, there is a green bar on the left side of the tile for "suggestions." Figure 3-3 shows

Figure 3-2: Old document UI featuring interaction between Jodie's document and Jonathan's


Transcript now
Jodie made changes to Westside Book Club

suggestions
5 entries

suggestions

Emily - Nov 11, 2015, 3:39 PM
War and Peace

Alex - Nov 11, 2015, 6:39 PM
Ready Player One

Jonathan - Nov 11, 2015, 8:39 PM
The Little Prince

Jonathan - Apr 7, 2:28 AM
Cracking the Coding Interview

Jodie - Apr 7, 2:29 AM
6.uar

each page as the user navigates through the document.

### 3.1.1 List of Social Datatypes

- *Lists (Submissions and Messages)*

  There are two types of lists with names "submissions" and "messages." Submissions and messages represent "conversations" where multiple users contribute in a structured way. They are lists of elements of the same type. Users may add entries into submissions or messages, and each entry of a submissions or messages is associated with a creation time and the user who created it. While messages do not request document subscribers to add in an entry, submissions remind users to add in an entry.

- *Poll*

  "Polls" allow users to vote from several options. Polls are a special type of submission list that allows only one entry per user and aggregates all the entries into results of the poll. Polls allow users to vote on one of the options and also change their vote. Polls do not display voting results until a user has voted.

- *Form*

  “Forms” are similar to physical forms and corrrespond to objects or records in standard programming languages. Fields are statically named, and each field may be any one of the components listed in this section. The fields in a form may be optional, possibly containing no value..

- *Choice*

  A “choice” has the same functionality as a drop-down menu or a multiselect. Users make a choice from a set of statically named fields. It is essentially a form that allows users to make a selection from the fields available and correspond to disjoint union types in functional programming languages.

- *Atom*

  “Atoms” are basic units of data. There are many different types of atoms such as Numbers, Units (checked/unchecked), Strings, Links (URLs), and Booleans (represented as yes/no). They can be writable or not writable.

  As the name atom implies, atoms are indivisible. This means that that they cannot contain other components other than their basic unit of data. Thus, atoms are always the leaves of the Transcript document tree as they do not contain children. They are often contained in other components – there can be messages with string atoms as entries, a form with number fields, etc.

## 3.2 Document UI Contributions

When initially designing the document UI, the Chorus designer was largely untouched – users were expected to create a document through a typed language. Thus, creating a document UI made Chorus similar to Hypercard in that it involved a simplified programming language and took away the need to design the UI from the user.

### 3.2.1 Displaying Tree Structured Documents

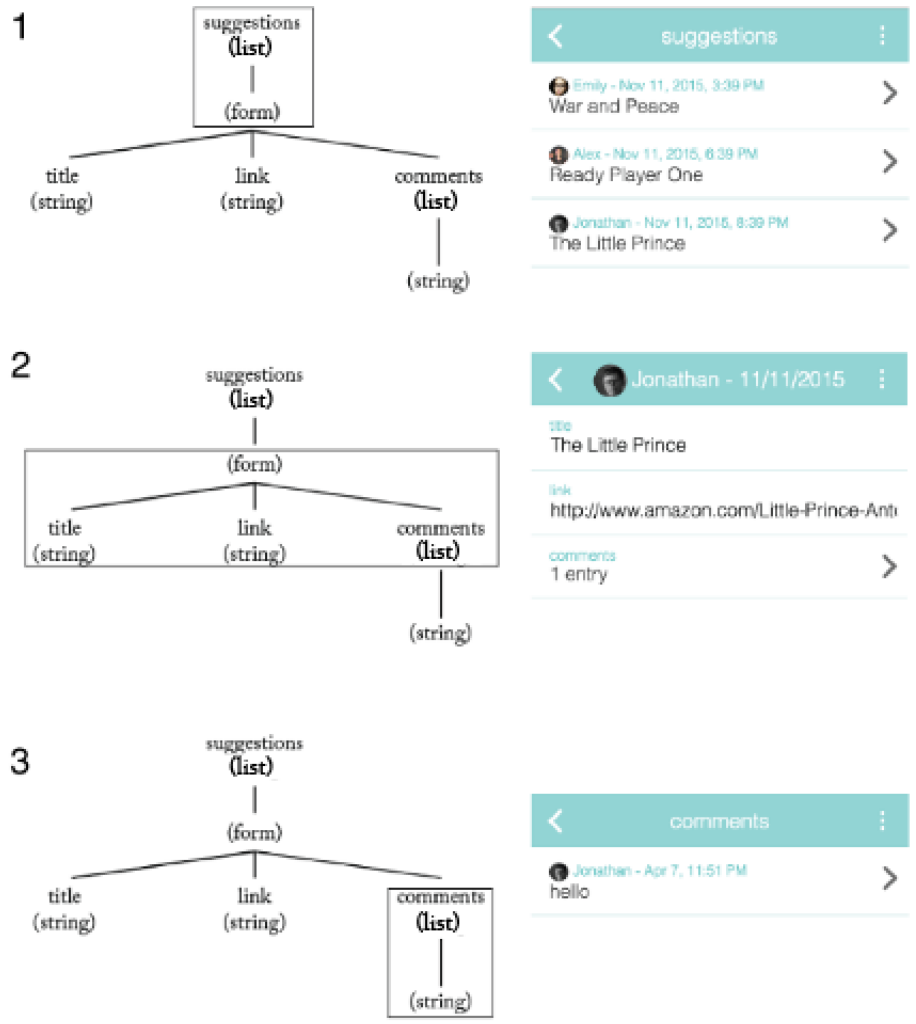
The first challenge we were faced with was how to convert navigating a tree structure into navigating a mobile UI. The most logical decision was to render each group of parent and children (two layers of the tree) as a page of the application. The parent would be the container of the page and display a summary of its contents, the children, of the parent. Clicking on a child summary when on the parent page would navigate to another page, using a horizontal scrolling animation, in which the child would be the container of the new page and a summary of its contents would be shown. We decided that each page should display summaries of the children instead of the children themselves because a child that is a parent of a subtree would too many layers and content on one page. Doing this preserved context as the parent and children would be rendered on the same page. Additionally, it allowed for navigation down the tree. A back button was used to navigate back up to the previous layer of the tree. Since atoms act as leaves of the document tree, atoms cannot be clicked to move down a layer. This design, shown in Figure 3-3, largely satisfied our needs, and it was slightly modified later to accommodate problems creating the designer UI brought up. These modifications are elaborated on in "Embedding and Breadcrumbs."

### 3.2.2 Displaying Social datatypes

In designing our navigation, we decided that each social datatype, apart from atoms, would have displays as both a container and a summary. This is because in the document tree, all components aside from atoms can have children. Thus, we had to design the displays for the datatypes as both containers and as summaries. Each datatype has two main pieces of information to display – its label and its content.

When a component is the container of the page, its content is already displayed, so we opted to make the label of the component into the header for the page. This helps provide context for the location inside the document. We expected each container to contain multiple child summaries, so we wanted component summaries to maintain a

Figure 3-3: The displayed document UI as the document tree is traversed.



suggestions

Emily - Nov 11, 2015, 3:39 PM
War and Peace

Alex - Nov 11, 2015, 6:39 PM
Ready Player One

Jonathan - Nov 11, 2015, 8:39 PM
The Little Prince

Jonathan - 11/11/2015

title
The Little Prince

link
http://www.amazon.com/Little-Prince-Ant

comments
1 entry

comments

Jonathan - Apr 7, 11:51 PM
hello

sense of conformity with each other but still remain separable. To maintain separation between component summaries, each part is bordered and represented as a tile in the UI. To maintain uniformity, each tile displays the component label and a summary of its contents. The summaries of content are different for each social datatype and serve to pinpoint the important data of each. For example, in Figure 3-3-1, on the suggestions page, the heading of the page is “suggestions” and the users can see the contents of the first field in each form, the children of “suggestions,” in the tiles.

### 3.2.3 Embedding Lists

As we created different Chorus applications, we noticed that many documents had a lot of levels in the document tree, making it hard to navigate. A lot of the problems seemed to be centered around list components. Viewing a list entry would often require a user to navigate through three separate levels – the container of the list, the list, and then finally the entry itself. After considering several of the applications of a list structure, we realized that in most cases, showing summaries of the list entries inside the container of the list offered no setbacks and made it easier to navigate and view the desired information. Thus, we decided to collapse list structures. When a list is contained in a form, summaries of the list entries are viewable in the form. Instead of going through three levels to view list entries, users now only have to go through two levels, condensing the height of the document tree and making it easier to navigate.

# Chapter 4

# Chorus Designer

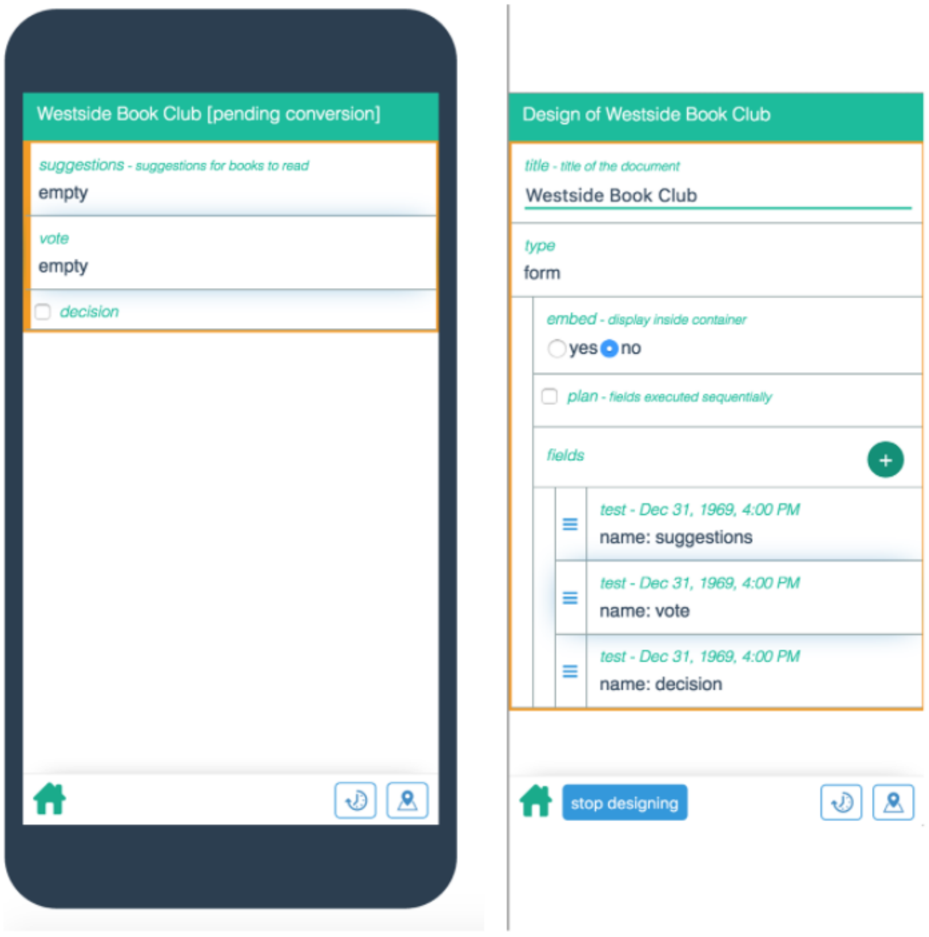
For the designer UI, we were tasked with even more problems to solve. We had to explore where we wanted the designer feature to be accessible from – trying out both a desktop application and as built into the mobile application. We also wanted to make a UI that was distinguishable from the normal document UI, but familiar enough so that document users do not have a large barrier of entry. Lastly, designing the UI also came with the task of gluing together the disjoint options users had in changing both the cosmetics and structure of the document.

Defining the requirements for the designer UI took many meetings and iterations. After showcasing our UI at workshop demos or presentations, we were able to receive helpful feedback on usability, which in turn affected our ideas for what we expected our application to be. Initially, we were focused on the idea of simplifying programming with a simple language, but decided to shift the focus to the user experience. For example, the designer portion of the application was just a text editor on the side and has now evolved into a WYSIWYG editor. The next two sections describe the actions that that can be taken in the designer and the contributions we made chronologically.

## 4.1 Designer Parts

The API for the designer is implemented so that an API call generates a "design document" and "converted document" from the backend representation of a collaborative

Figure 4-1: The initial converted document (left) and metamodel designer (right)


Westside Book Club [pending conversion]

suggestions - suggestions for books to read
empty

vote
empty

decision

Design of Westside Book Club

title - title of the document
Westside Book Club

type
form

embed - display inside container
yes [✓ no]

plan - fields executed sequentially

fields

test - Dec 31, 1969, 4:00 PM
name: suggestions

test - Dec 31, 1969, 4:00 PM
name: vote

test - Dec 31, 1969, 4:00 PM
name: decision

document. From there, the design document has an additional set of API calls that allows users to perform various actions that affect the display and structure of the collaborative document. The converted document is the updated collaborative document – all the structural and display changes are present in the converted document before the changes are published by the user. When changes made using the design document API are saved, the converted document become the new collaborative document. Figure 4-1 is an example of the old metamodel designer where the pane on the right side is a visual representation of the design document and the phone on the left side displays the converted document. Interactions with the fields in the design document alter the displayed converted document on the left side. For example, using the drag handles on the left side of the tiles in "fields" to reorder the items in the list will reorder the corresponding tiles in the converted document.

The designer API allows changes to be made to both the structure of the document as well as the display. The following section lists the changes the design document supports.

### 4.1.1 List of Designer Actions

- *Adding a Field*

  New fields may be added to form, choice, and poll components as they can contain multiple children of a different type. Attempting to insert a new field into any other component will not work. The root of the document tree is always a form.

- *Removing a Part*

  Any component may be removed from the document. Removing a parent of a subtree will also remove all nodes of the subtree. In the case where list components, like submissions or messages, are a parent of a subtree, removing a field from the subtree removes the field from all the entries inside the list component.

- *Moving a Part*

  Any component can be moved down a level of the tree with two methods. A component may be wrapped in a form, in which case the component becomes a field in a form. A component may also be wrapped in a list, in which case entries in the list will be of the component.

- *Renaming a Part*

  Each component consists of a title and stores data. For components other than the entries of lists, the title may be changed.

- *Changing Part Type*

The type of a component may be changed to compatible types, i.e. from a messages to submissions or from one atom to another.

- *New Page*

  Each page in the Chorus document can display multiple layers of the document tree, a design decision explained in the Contributions Chapter. By default, each page in the Chorus document corresponds to two layers of the document tree structure. Each page displays the parent and summaries of all its children. Users have the option to choose whether the children's children are also displayed together, in which case each the children's children's summaries are displayed. Since atoms do not contain any children, atoms are always displayed on the same page as their parent.

- *Required*

  Making a field required lets the document know to put an error state on the field if the field is empty.

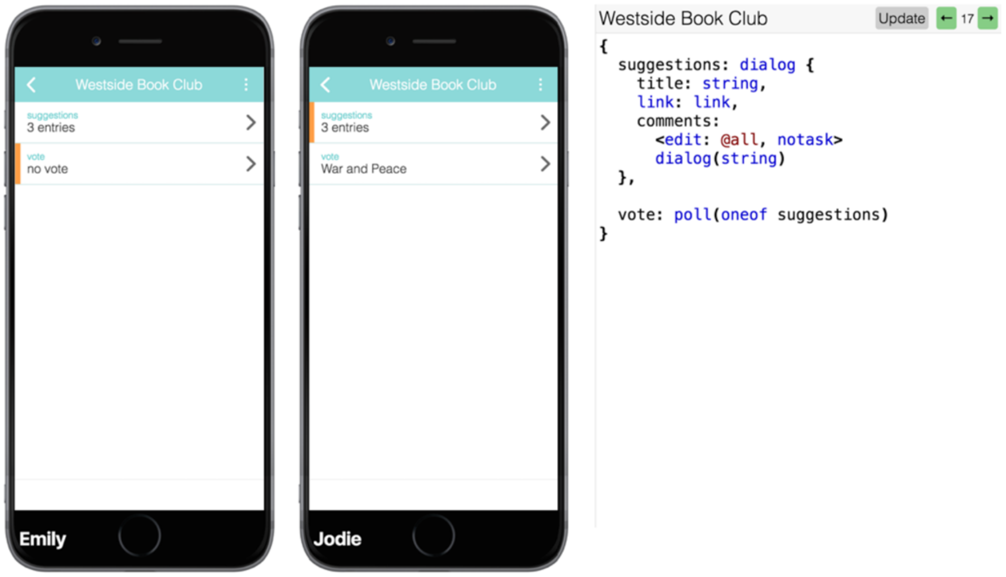
## 4.2 Designer UI Contributions

After creating a working UI for the document, we decided to move on from creating a document with a typed language to creating one with an application. Figure 4-2 is a screenshot from our designer before transitioning away from a programming language. The pane on the right contains code that describes the documents displayed on the left side. We explored multiple ideas for how the designer could be implemented, from a separate "metadocument" to embedding the "metadocument" and finally moving to a WYSIWYG model for the designer. Along the way, we made several changes to the document UI as well to better cater to the needs of the designer.

### 4.2.1 Metamodel Designer

We played around with the idea of using a "metadocument" for the design, which will also be referred to as a "design document." The design document would be rendered

Figure 4-2: Old designer with code editor.



```
{
  suggestions: dialog {
    title: string,
    link: link,
    comments:
      <edit: @all, notask>
      dialog(string)
  },

  vote: poll(oneof suggestions)
}
```

side by side with the collaborative document to help give context to the part being designed. With this approach, we intended the designer to be a collaborative document itself. Each component in the collaborative document being designed is associated with multiple, hardcoded components in the design document. For example, a form in the collaborative document may be associated with a choice for its type (allowing the user to change it from a form to another component), a list of forms for its fields (adding an entry into a list adds a new field to the form, and will also allow the user to change the type for the field), and so on. The design document is a metamodel of a collaborative document as it contains data and a hard-coded structure based on the collaborative document that is currently being designed. Changes to data in the design document change the design of the collaborative document. This is shown in Figure 4-1.

Since the design document is a hardcoded collaborative document, no extra code is needed to render it. The added benefit of this approach is there are no additional affordances needed to make this easy-to-use for the user because the designer shares a UI with the collaborative document. However, since each component of the collaborative document is associated with multiple components in the metamodel of the document, this multiplicatively increases the layers of the tree, making it harder

Table 4.1: Metamodel vs. WYSIWYG

| Pros/Cons | Meta-model | WYSIWYG |
|---|---|---|
| No New Affordances | yes | no |
| Familiarity from Previous Applications | no | yes |
| Less UI Code | yes | no |

for users to navigate. There also exists a disconnect in how actions in the design document affect the collaborative document because the changes do not occur in the same space. Unfortunately, after prototyping the metamodel designer, we realized the negatives of its complexity outweighed the benefits of UI familiarity.

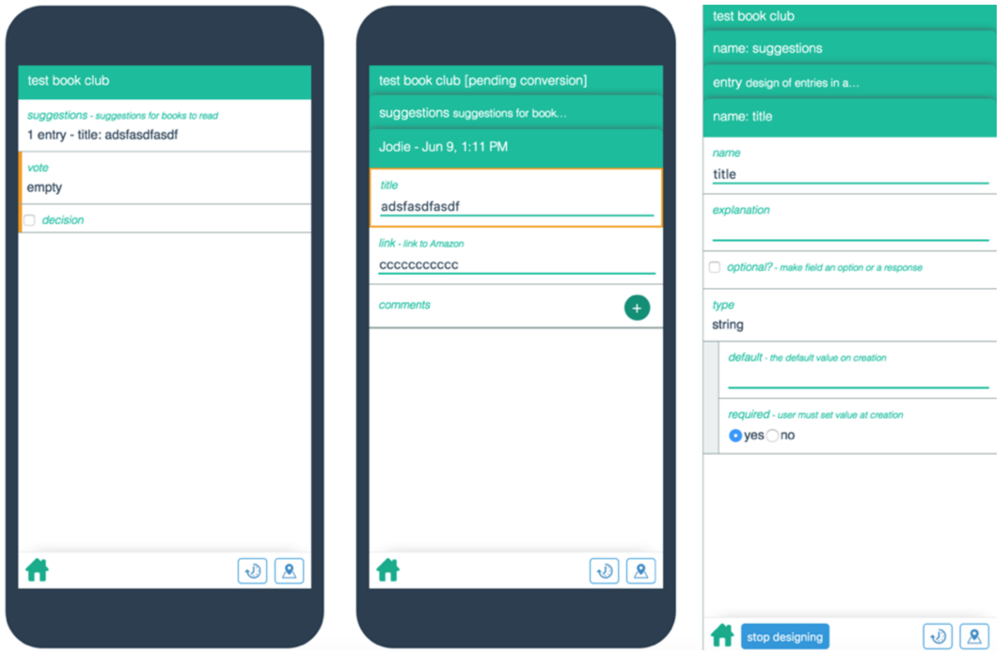
### 4.2.2 Embedding and Breadcrumbs

The design of each component in the document is represented as many multiple nested components in the document's metamodel. Only two levels of the tree are shown in our document display, which led to many usability problems. There were many levels to drill down to, making it easy to get lost inside the document. Additionally, the multiple components in the document's metamodel that correspond to the design of one component are on separate pages, making it impossible to design all parts of the component at once.

To fix this problem, we came up with the idea of embedding. Instead of each page only showing the parent and its children, each component has an embed option. If a component is embedded, then its contents are shown on the same page as its parent. This allows many levels of the tree be shown at once and the option of showing multiple levels of one subtree under a node and only one level of another. Only non-embedded components will be the parent in a page.

To give more navigational context and allowing users to traverse back more than one level of the tree at once, we implemented vertical breadcrumbs. These manifest as titles of cards stacked on top of each other – clicking one of the titles brings the user back to the page with the title. This replaced the horizontal sliding animation we had earlier. To prevent the breadcrumbs from taking up too much vertical space, the maximum number of breadcrumbs that can be shown is four. Only non-embedded

Figure 4-3: A screenshot taken prior to the redesign in Section 4.5.2 that showcases embedding (on the right hand pane) and vertical breadcrumbs.


test book club

suggestions - suggestions for books to read

1 entry - title: adsfasdfasdf

vote

empty

decision

test book club [pending conversion]

suggestions suggestions for book...

Jodie - Jun 9, 1:11 PM

title

adsfasdfasdf

link - link to Amazon

ccccccccccc

comments

test book club

name: suggestions

entry design of entries in a...

name: title

name

title

explanation

optional? - make field an option or a response

type

string

default - the default value on creation

required - user must set value at creation

yes [✓] no

components will show up in the breadcrumbs.

### 4.2.3 Designer Location

One of the biggest criticisms in usability we received from presenting the metamodel designer was that there was too big a disconnect between the component being designed and the designer itself. The disconnect was largely due to the document and designer being rendered in separate mobile-phone sized spaces. This led to us experimenting with a mobile designer embedded inside the document. We created several mockups, shown in Figure 4-4, and prototyped the two most promising ideas: a design panel on a new page and a design panel under the component being designed. After playing around with both options, we concluded that an embedded design panel felt the most natural. Since document design changes are not expected to be the bulk of the interactions in the Chorus application, we opted to add in a button that toggles the user between "design mode" and non-design mode instead of having designer options always-on. Thus, to make design changes, users toggle the design button in the document button bar and then go through the design panel. During this prototyping, the metamodel was largely simplified as users no longer needed navigate through the design document and several metamodel tiles were condensed into buttons. As we moved away from the metamodel designer, we no longer had to display very complex structures, allowing the document UI to evolve into a cleaner, more “document” look. Moreover, the integrated designer no longer required the whole vertical length of the phone screen, making the design panel feasible.

### 4.2.4 WYSIWYG

While integrating the designer into the document solved some usability problems, there was still room for improvement. For example, to modify the label of a component, users had to do so through an input in the design panel as opposed to directly modifying the label itself. To improve usability, we found options in the panel that could be translated to user interactions. Drag-and-drop actions were implemented and replaced buttons for moving components around and adding in new components. Component labels were changed to offer editing in place. The type of the component was also added as a smaller label to each component.

### 4.2.5 Redesign

Inspired by Notion’s minimal UI and the desire to make the Chorus document more reminiscent of other collaborative workspaces like Google Documents, we decided to redesign the Chorus document to be more minimal and textual. Doing so brings more focus to the contents within the document.

### 4.2.6 Minimap

Due to the highly nested nature of Chorus documents, we wanted to create a way for users to easily traverse the document without having to go page by page. Thus, a Sublime Text Editor inspired minimap came to be. In the minimap, each component in the document is represented as a box that looks like a tile (Figure 4-6). Tiles in the

Figure 4-4: Mockups of potential designer locations and interactions done in Adobe Illustrator.

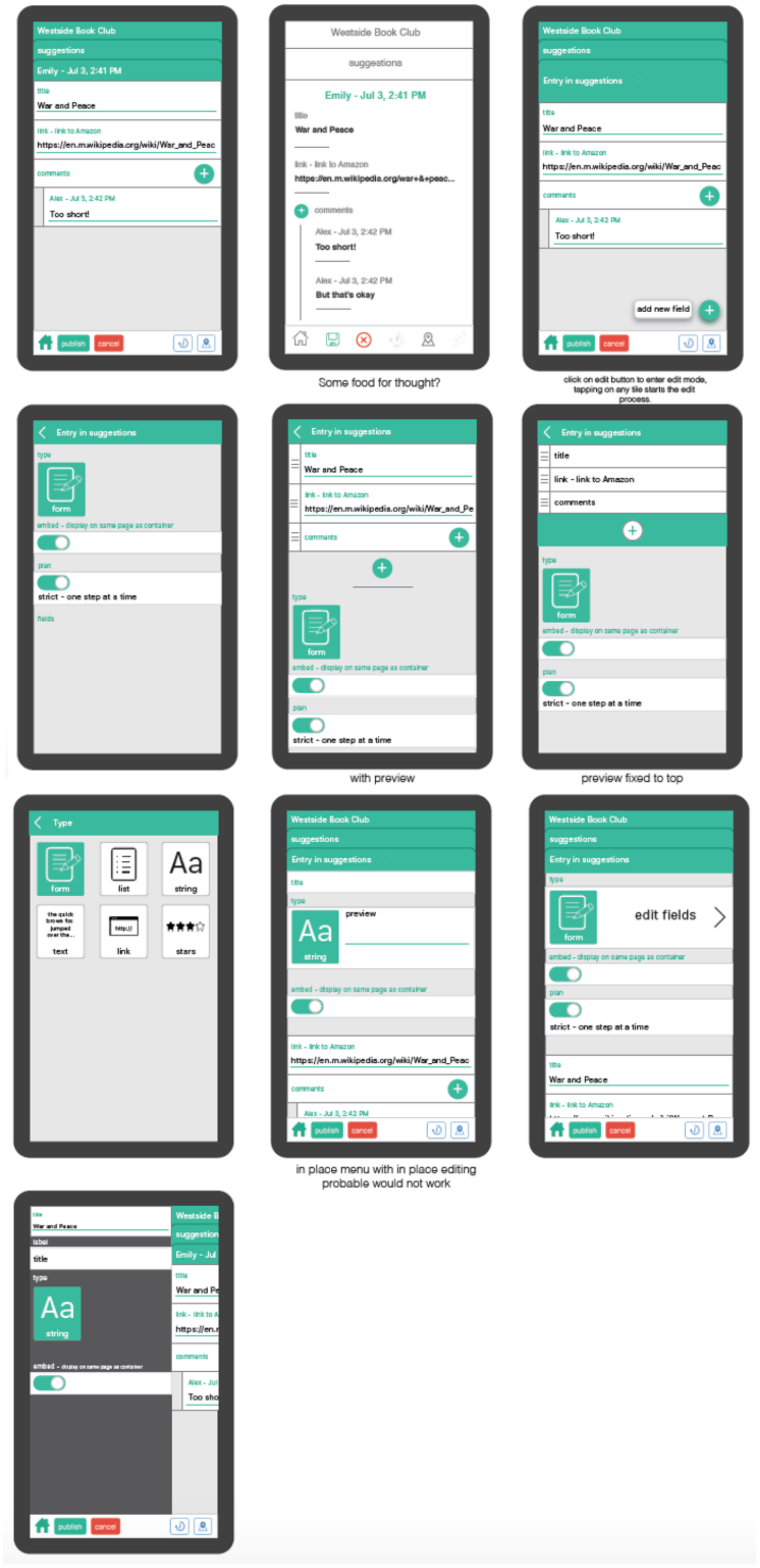
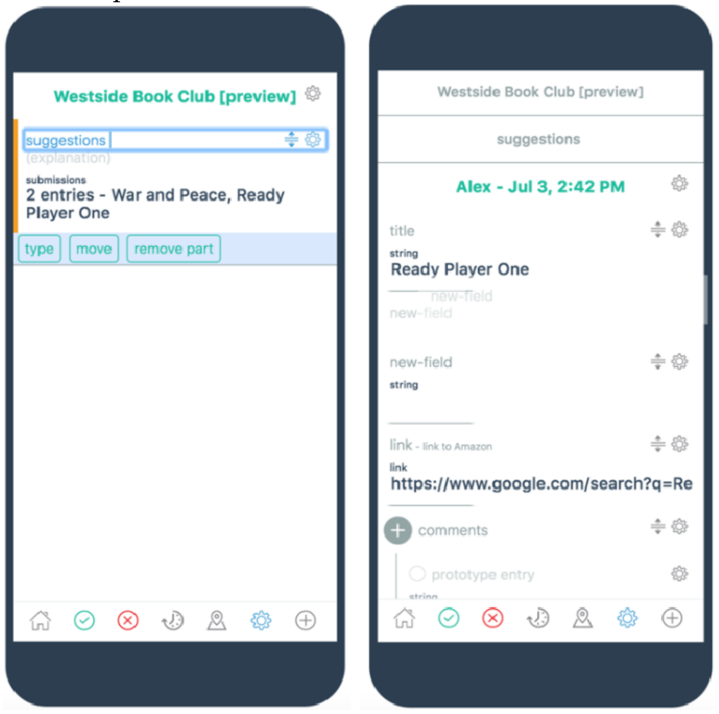
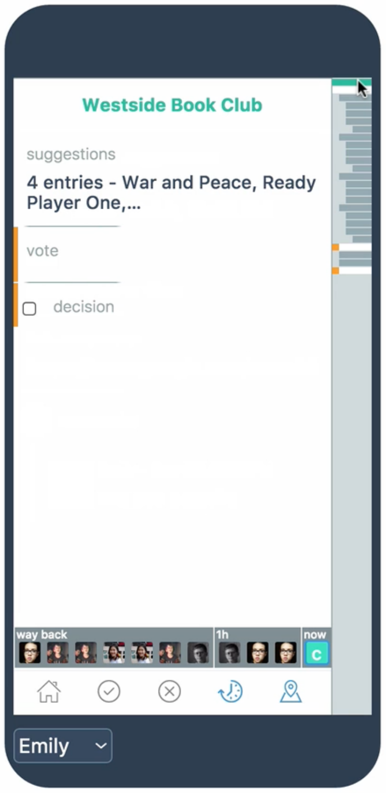


Figure 4-5: WYSIWYG designer. Left side shows in-place-editing of labels and right side shows drag-and-drop interaction to add in new fields.



minimap are colored gray when they are not in display on the current page. Embedded components in the minimap are shown by indenting the tiles. When clicking on a box in the minimap, the user is taken to the component that is represented by the box. This feature is especially useful in conjunction with component states – component states show up in the minimap as colored bars by the boxes. For example, a newly edited component will have a blue bar next to it on the minimap. This means that a user can quickly find and navigate through the document to the newly edited component through the minimap by seeking out colored tiles on the minimap.

Figure 4-6: The minimap on the right side consolidates notifications on states and helps navigation



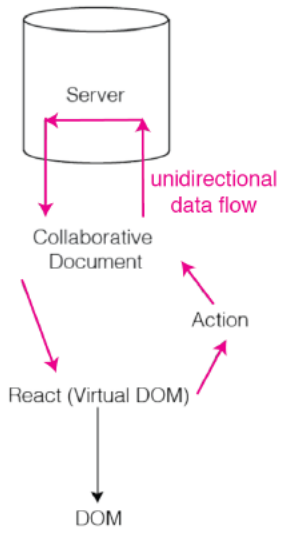
# Chapter 5

# Implementation

Our technical approach to Chorus consisted of multiple decisions. We decided to implement our desktop application as a single page Electron application in order to rerender the display of content easily. The rerendering of the UI happens with changes to the document structure. When a user interacts with the Chorus document or designer, the user will changes the document state or the design document state, which alters the structure of the collaborative document. The UI needs to rerender the changed data in the design document and the changed structure of the collaborative document. Thus, in order to rerender the display of the content easily, we wanted the UI framework to automatically manage all the Document Object Model (DOM) updates when the underlying data changes. Single page applications load a single HTML page and do not require constant reloading of the page or communication with the server, making the application more responsive. An additional benefit of implementing the web application as a single page application is that the web application can be run offline for presentations during conferences and workshops.

The UI was built with React.js, a Javascript library for building user interfaces, as a layer on top of a collaborative document model. React best fits the structure of and the flow of data in the design document. As mentioned previously, the structure of the design and collaborative documents are based on statically typed trees. We considered routing based frameworks where the state of an application is represented by a URL, with each URL corresponding to a specific display. However, since the

Figure 5-1: Data flow in our implementation of Chorus.



display on screen depends on the location of the user in the tree of the document and there are many possible trees, a routing based framework did not make sense conceptually as each document tree would have its own unique set of URLs.

Additionally, the flow of data from the user's design document to the structure of the collaborative document stored on the server and then to the client's copy of the collaborative document to the DOM closely mimics a unidirectional data flow. This mimics the Flux application architecture, which eschews MVC in favor of a unidirectional data flow (Figure 5-1). Flux application architecture and unidirectional data flow complement React's declarative style and makes it easier to track changes [4]. Our model closely follows this pattern, solidifying our choice in React.

Choosing to use React also has additional benefits. Its automatic UI rerendering is very useful for displaying the underlying document as there is no extra processing needed to be done on our end. React is essentially a virtual DOM and is faster than other Javascript frameworks. When the underlying data changes, it efficiently recalculates the differences in the DOM to avoid rerendering every node, making it faster at rerendering applications than other frameworks [10]. We also plan to eventually port our application to the mobile platform. React Native, which allows developers to write native mobile application with React, will greatly help simplify

Table 5.1: React vs Other Routing Based JS Frameworks vs Multi Page Applications

| Requirements | React | Other Frameworks | Multi Page App |
|---|---|---|---|
| No Server Required | yes | yes | no |
| Good Offline Capability | yes | yes | no |
| Less Dependence on Document Structure | yes | no | yes |
| Unidirectional Data Flow | yes | no | no |
| Easily Port to Mobile Application | yes | no | no |

this process. A summary of the pros and cons are listed in Table 5.1. The next two sections describe interesting React implementations.

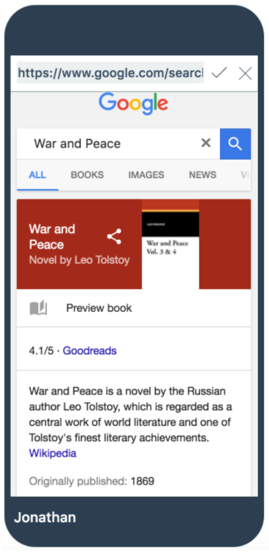
## 5.1 Page Transitions

Prior to reorganizing the layout of the document into vertical breadcrumbs, opening a new page in the document would result in a horizontal sliding animation from the old page to the new page. The horizontal sliding animation was added to make the prototype more realistic for demos and presentations. Though it sounds simple, the sliding animation required a few workarounds because we were rerendering pages with React. When a user clicks to open a new page, the backend puts forth the data of the new page and the data of the previous page is lost. There is no way for React to render the previous page for the transition as there is no data from the previous page to render. To combat this, on a page transition before the render of the new data, we translate the React components currently on the page to the HyperText Markup Language (HTML) tags viewable on the Document Object Model (DOM) and save the translated React components. Thus, when simulating the sliding movement, we can render both the previous page, by inserting the saved HTML components, and the new page, with React.

## 5.2 Captive Browser

One of the features of the Chorus document is a captive browser. Clicking a link component opens up a browser within the application. Initially, to simulate this for

Figure 5-2: Captive browser in the redesigned (Section 4.2.5) document UI



War and Peace
Novel by Leo Tolstoy

4.1/5 · Goodreads

War and Peace is a novel by the Russian author Leo Tolstoy, which is regarded as a central work of world literature and one of Tolstoy's finest literary achievements. Wikipedia

Originally published: 1869

demos and presentations, we created a React iframe component and turned browser web security off in order to load contents from foreign pages. However, turning off web security is not optimal so we looked toward other solutions. We turned to Electron, a framework that allows desktop applications to be made using HTML and Javascript. Not only did Electron let us make Chorus a standalone application, but it also has a built in webview tag, which embeds guest content. Unlike an iframe, webviews run asynchronously from the app that embeds it, nullifying the security issues that iframes have. The captive browser in Chorus is made up of a React webview component and an input. Navigating to a site updates the input box with the URL of the site. Changing the contents in the input and pressing enter also points the webview to the URL entered.

# Chapter 6

# Results

## 6.1 User Testing Setup

To assess the learnability and user experience of the collaborative document UI and the designer UI, we performed two sets of user testing using the thinking aloud protocol (n = 1 for the collaborative document and n = 3 for designer). The thinking aloud protocol is a usability method that gives test subjects a set of representative tasks to perform and asks them to voice out loud what is going on inside their head as they perform them. It is a commonly used study that is useful for pinpointing misconceptions in the UI that potential users have. The protocol is also known for being robust, so mistakes in conducting the user tests do not greatly impact the validity of the results.

The thinking aloud protocol was chosen for how well it fit our goals from the user test as well as for its cost efficiency. Our goals for user testing the UIs for the collaborative document and the designer were to get a robust sense of large missteps in both designer and document UIs and general areas to improve. As such, the downsides of a thinking aloud study were not a impactful. In the next sections of the chapter, we summarize the conclusions drawn from each user test. The script for both user tests can be found in the Appendix One.

From both of our document user test and designer user tests, we learned that the conceptual idea of Chorus was easy for users to pick up. At the end of each test, all the subjects could answer the conceptual questions correctly. We also learned that most of the UI problems were discoverability related. Subjects were clicking around the UI a lot, but still had trouble realizing that something could be clicked or that other actions, like dragging, could also be performed.

## 6.2 Document User Test

A user test on the Chorus document UI was performed to address severe problem areas in the UI before moving to testing the designer UI. The subject was asked to perform tasks that assessed the learnability of the various buttons in the UI, modifying document contents, and the navigation in the document.

### 6.2.1 Document Test Subject One

A thinking aloud study was done on a female, nonprogrammer by a neutral third party. Throughout the study, the neutral third party asked the participant to perform certain tasks in the Chorus application, and in between tasks, asked general questions to assess the participants understanding of the user interface.

From watching the participant interact with the product, we were able to make several observations about the problem areas in the user interface.

The structural representation of the model was unclear. The participant had trouble navigating between pages. They discovered the minimap, but were unable to connect the boxes in the minimap to tiles in the UI. There might have been too many boxes in the minimap to understand the relation between the two. Additionally, there was some confusion in navigating between pages. It was unclear at first on how to open up a new page, and even more confusing in how to navigate back up the document. To alleviate this, we added an upward pointing caret icon in the breadcrumbs to make it more obvious that they can be clicked to navigate through the document. For designer testing, we disabled the minimap and timeline features as they are advanced user features that are meant to be enabled through an additional menu.

We also found the colors were not helpful indicators of the status of the document. The participant did not really pay attention to them during the tasks. After the study was finished, the user did not notice the correct meaning behind any of the colors. The colored bars need to be more obvious, and there might be too many colors for users to understand. This can be ameliorated in the future by some sort of onboarding process – it was not changed for the designer UI testing as it was irrelevant to the tasks we asked the users to perform.

Some icons were not obvious enough in their meaning. The buttons were somewhat unclear in their meaning. Reusing the checkmark icon for multiple purposes resulted in the participant not understanding the ”commit” button. The minimap and timeline buttons were changed to be hidden as they are more complicated user interactions that offer additional actions that may confuse non power users. To combat the confusion, the checkmark icon is explained in the onboarding screencast for the designer testing.

The flow of interactions was hard to understand. The autofill for the link caused a little bit of confusion at first. Additionally, the “multi action then commit” model was also a little confusing for the participant – the reason is more likely because of the icon confusion though. The interactions may be improved if the color usage were better. However, this was not addressed for the designer testing.

## 6.3 Designer User Test

In the designer user tests, users were given a brief introduction video to watch, which explains how to create a simple document that consists of a list of forms. Users were then asked to perform a set of representative tasks: adding in a new field, changing component type, moving components, and changing options. With each test iteration, we were able to make improvements to the UI that decreased the time required to complete the set of tasks. While task completion time is affected by how much attention subjects paid to the introduction video, nevertheless, the continued improvement on task completion time and conceptual understanding through each iteration show that our changes were successful. Since each test subject was able to successfully complete the set of tasks, we consider our designer UI a success.

### 6.3.1 Designer Test Subject One

The first subject for a user study on the Chorus designer is a female, nonprogrammer whose second language is English. The subject took 38 minutes to complete all the tasks.

From the observations made during the test, we made a few conclusions on which changes were necessary to make prior to the second iteration of the designer testing.

First, the move button is confusing, and was removed in the designer as the actions performed by the button are more for power users. Additionally, the name of the button is confusing as the most common "moving" a user will do is toggling on or off the embedding. It took a long time for the subject to figure out the correct way to toggle the embed. This can be explained away by the language barrier. As the subject is a nonnative English speaker, she stated that she was confused about the meaning of embed and thought embed to have an opposite meaning. However, we renamed "embed" into "new page," as even without a language barrier the word "embed" is not clear in what it does.

The participant is also confused by the home button, it is unclear that home takes the user to all of the documents rather than the root of the document. This issue can be remedied by showing how to move up the document in the introduction video. However, this is not an issue tied in the designer so no changes were made. The subject was asked to navigate back to the root of the document without clicking home, and was able to figure out how to. This demonstrates that the changes made following the user testing on the user interface were useful.

The subject had to be reminded what the first task was after completing part of it. The first task requires a large number of actions, which may take a long time and can also be confusing. Instead, the first task should be shorter and broken down into several parts. Instead of telling subjects to add in a field called "comments," which accepts comments from multiple people, the task should be broken down into telling subjects to (1) add in a new field and name it "comments," (2) make comments a list that any user can submit to.

At some points, the subject was also confused about the difference beteween design mode and not design mode and spent a large amount of time not in design mode. This should not be an issue in future tests as it can be attributed to the language barrier – the subject thought that the title "suggestions" meant "suggested design." We decided if that issue occurs again, then we should add textual indicator in the title for when the user is in design mode.

The plus button next to the "suggestions" title was confusing as it added a new entry into "suggestions" but was not a designer related action. This confused the subject between adding a new field to the structure and adding a new entry. To fix this confusion, the plus button is disabled in design mode for the next user test so users cannot add new entries.

The click and drag to add a new field was also a confusing mechanic. It was shown briefly in the introduction video without explanation. To remedy this, the introduction video was made to include a verbal explanation for how to add a new field.

Lastly, the subject commented that the application might become too "crazy" if everyone comments and there should be limits to how much information can in an application or a page of the application. We have considered limiting the number of fields or items shown per a page before, and have decided against it. This was not addressed, but could have been by making new items automatically not embedded to limit thing shown per a page.

### 6.3.2 Designer Test Subject Two

Subject two is a male nonprogrammer. At the beginning of the test, he is briefed about the thinking aloud protocol and watches an updated introduction video before starting on the tasks. Unfortunately, during this test, changing between some incompatible types resulted in bugs that could not be resolved during the user study, forcing the test to be restarted in between the first and second task. All of the actions in the test took around 27 minutes to complete including the bugs.

From this user test, we learned that the discoverability of the type menu needed work. For this subject, changing a field type and finding the correct type was the task that took the longest to complete. To combat this, the type menu was shortened down to the social datatypes that currently are supported for the next iteration of the test. The type menu descriptions were changed to be a little bit more descriptive as well. Instead of displaying the type (which opens up the type menu) as small black text, it was changed to have blue text and border to have a more button look.

Changes are not automatically saved when a user leaves design mode which was confusing for the subject, though the subject was able to recognize that his changes were not saved and fixed it for the second completion of the task. In the future, it might be better to have changes remain pending so that if a user leaves design mode and reenters it, the pending changes are still there. However, this was not pursued in the next iteration of the designer interface.

The add entry button disabling for lists in design mode was a little confusing. But, the subject did not confuse it with the add field button, so the disabling was effective in that sense.

The subject indicated a desire to be able to add unique fields for entries in a list – this feature could technically be satisfied with "optional" or "script."

The subject also indicated that the names of type and descriptions could be clearer. In particular, he found the "form" name to be a little bit deceiving as form is more tied to options whereas the "form" in the application is more of a container for multiple fields. Unfortunately, the subject was not able to come up with a better name. The descriptions for each type were changed for the next iteration of the test to make the type names clearer.

With the discoverability and self-explanation of "new page" option, the explanation was removed from the video to see if users can figure it out for themselves.

### 6.3.3 Designer Test Subject Three

Subject three is a non-computer science major with some experience programming. No further improvements were made after this test, and possible improvements will be left to future work. The test took around 23 minutes to complete, a large improvement from the second test.

The subject was a little bit confused about the "new page" toggle at first and whether it could be toggled on infinitely. A solution for that would be to remove it for atoms and only have the option there for containers.

Since the subject did not have any trouble changing the type of the "comments" component, the changes we made after the previous test were helpful. Also, the subject was able to find and use the "new page" option without any explanation from the video.

Similar to the second subject, the third subject had difficulty remembering to click the gear icon to edit the name in multiple times throughout the test.

The subject had some trouble in differentiating between the add button on top in design mode and not in design mode. Thus, it might actually be beneficial to reenable the add entry button in design mode in the future.

The subject mentioned that the color choice could be better as it was too minimal in design mode. He implied that it is confusing to which buttons can be clicked and which buttons are disabled because the scheme for what colored buttons mean is inconsistent with each other. Maybe there should be a difference between colored outlines and colored shapes, or simply not show the buttons that are actually disabled. He also mentioned that there should be colors to let people know what was changed.

Because of the significant improvement in completion time from the initial designer test, no further changes were made to the designer. However, the things mentioned above will be helpful to implement in future iterations of the designer.

# Chapter 7

# Conclusion

In this work, I explore the background of Chorus application and my research in designing a UI for the Chorus document and the Chorus designer. A successful standalone desktop application was made that simulates user interactions with a future mobile application. User tests were done, and confirmed that the UI is successful in allowing users to learn and perform representative tasks within a short period of time – under 25 minutes. This research on Chorus has been presented at the 2016 YOW! Conference and at a workshop in ECOOP 2016.

Future work can be done on increasing the social datatypes we support in the application and improving certain aspects that we've observed from our user studies. Additionally, it will necessary in the future to prototype on a mobile platform, potentially porting over with the use of React Native and further user tests need to be performed.

The past 1.5 years spent on this project have been a great learning experience. I learned a lot from all steps of the process, from designing the very first iteration of Chorus to conducting the user studies. One of the most surprising things is how many design iterations we went through and how much time Jonathan and I spent at the whiteboard drawing and discussing ideas. The biggest roadblocks were visual design based, not internal code design based. Particularly interesting is that more than once, we'd reason and decide upon a way to display something in the UI, but after prototyping, change our minds for an option we eschewed. For example, we originally believed that a desktop designer would fare better than a mobile designer, but changed our minds once we played around with a desktop prototype. Even though we put a lot of thought into each design we implemented, the idea did not always translate successfully. Thus, it was important to learn to throw away useless code and start from scratch.

When performing user tests, a few situations threw me off guard. I simultaneously underestimated our test subjects and overestimated them in certain regards. For the document user test, our test subject navigated through the document using the minimap instead of through the vertical breadcrumbs. Both Jonathan and I consider minimap usage to be a very advanced tactic, and were impressed that the subject not only discovered the feature, but was able to use it quite successfully in the brief time it took to perform the test. On the other hand, some of the test subjects did not notice the colored bars we had in the UI and two explicitly stated that they were confused about the colors in the UI. We initially thought that the colors in the UI were pretty self-explanatory, so we did not put much thought on them. Additionally, for our designer tests, I was afraid that the introduction video would give too much information away on how to complete the tasks. Surprisingly, no one paid attention to the full video despite it being around two and a half minutes long. Each test, a subject would be completely confused about how to complete at least one of the tasks that was shown in the video. Fortunately, they would always figure out how to complete the task, indicating that the UI had good discoverability. From the disparity between the user test results and my expectations, I learned not to make any assumptions about which parts users have an easy or hard time understanding.

Finally, one of the most important pieces of day-to-day knowledge I picked up is of the power of deadlines. At times, progress on Chorus would stagnate and it seemed like none of the design ideas Jonathan and I came up with would work as well as they sounded on paper. However, deadlines for papers, workshops, and presentations interspersed throughout provided a constant motivation to make Chorus ready. In fact, Jonathan would sometimes schedule these events so that we had a timeframe to reach a certain goal. It worked well, and I would estimate around over a third of the work done on Chorus was in the two or three weeks before a deadline.

I am thankful for the opportunity to work on Chorus and the opportunity to be the mastermind behind much of the design. I believe Chorus has the potential to be a powerful tool in any social situation, whether it be for organizing teams in workplaces or organizing meetings and tasks in student clubs. I look forward to the future evolutions and seeing where Chorus ends up.

# Appendix A

# User Testing Script

The text in quotes was read out loud for the test subjects to hear. The introduction (A.1) was read aloud at the beginning of each study, and then either tasks from User Interface Testing or Designer Testing were given to subjects to perform.

## A.1 Introduction

"Thank you so much for coming into the lab today. Today we'd like to show you an application someone in the lab has been developing. We'd like you to help us figure out where some of the problems are and some of the things the application is good for. We are not testing you, so do not be afraid if you are confused, in fact, it will be very beneficial if you were to let us know if you have trouble figuring something out. The test we are doing today is something called the thinking aloud protocol. As you are using the application, do the best you can to voice the things that are going on inside your head. For example, if something were confusing, you can let me know that it is confusing and if you can, also why it is confusing."

## A.2 Document Testing

"Today I am going to put you into a scenario in which you are going to use this app. The scenario is that some of your friends from your book club have recently started using an app to help them organize their book club and want you to download it as well to participate. You have just downloaded the app, and you are now trying to participate in the discussion."

The following list corresponds to actions the user goes through in order to complete the task.

- Add a book suggestion to "suggestions."
- Save their suggestion.
- Comment on Alex's book suggestion.
- The test conductor then adds a comment on one of the suggestions and asks the participant to find the new comment.
- Vote for a book in the poll.
- Ask the meaning of the different colors and buttons in the user interface.

## A.3 Designer Testing

"Today I am going to put you into a scenario in which you are going to use this app. The scenario is that you are a co-organizer of a book club, and you have recently downloaded this app to help you organize your book club. The app allows you to design a custom mobile app for your club. Your co-organizer has already started designing the app, and the group members have already started using it. In order to get you up to speed, your co-organizer shows you how to add a new item in the app."

The user is then shown a brief video introducing them to the designer, and is asked to complete the following list of tasks.

- Add another item into "suggestions" and name it "comments."
- Change "comments" so that multiple people can add in their comment.
- Ask user if "comments" is also present in other suggestions. Ask why or why not.

- Reorganize the order of things in suggestions so that comments come before the Amazon link.
- Delete the link field.
- Display the suggestions and the person who suggested them within the first page, similar to how you can see comments without having to click anything.

# Appendix B

# List of States

### Writeable/Unwriteable

Parts may be writeable or unwritable by a user. For example, a collaborative document can be structured such that the fields of a form can only be “writeable,” more accurately editable in this case, by the user who published the form.

### Pending

Users may make multiple changes to the collaborative document. When they finish changes, publishing the changes pushes the changes to all the users subscribed to the document. “Pending” is intermediary state between publishing a changes and making changes, and is a bit that is set when a leaf of the document has been modified. The pending state is propagated upwards through the parent nodes until the root of the tree.

### Error

An error lets the user know when there is something wrong has occurred in a component. The collaborative document throws an error when the type of input by the user does not match the type defined in the schema of the document. For example, this may happen when users try to write a string in a number field or if the user does not submit a value into a non-optional field.

### Notifications

Notifications let the user know when there are new changes to the document (after a user publishes their changes) or when there is a todo. Notifications are set on the leaves of the document tree and are displayed on all the parent nodes until the root of the document. When the leaf the notification bit is set on is viewed, the notification bit is cleared.

# Bibliography

[1] "Backbone.js," http://backbonejs.org/

[2] J. Edwards, "Subtext: Uncovering the Simplicity of Programming," *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 2005

[3] J. Edwards, "Transcript: End-User Programming of Social Apps," *YOW!*, 2015

[4] "Flux Overview," https://facebook.github.io/flux/docs/overview.html

[5] "Docs, Wikis, Tasks. Seamlessly in One," https://www.notion.so/

[6] "JSX in Depth," https://facebook.github.io/react/docs/jsx-in-depth.html

[7] "The All-in-One Collaboration Platform," https://try.airtable.com/

[8] J. Nielsen et al., "The learnability of HyperCard as an object-oriented programming system," *Behaviour and Information Technology*, 1991, pp. 111-120

[9] "Ohm", https://github.com/cdglabs/ohm

[10] "React," https://facebook.github.io/react/index.html

[11] "React Native," https://facebook.github.io/react-native/

[12] "Routing," https://guides.emberjs.com/v1.10.0/routing/

[13] M. Wasson, "ASP.NET - Single-Page Applications: Build Modern, Responsive Web Apps with ASP.NET," *MSDN Magazine*, 2013, pp. 40-42,