Iterative User-Centred Design of Family Tree Photo Software

Michael Inglis

4th Year Dissertation
Software Engineering
School of Informatics
University of Edinburgh

2016

Abstract

The aim of this project is to create a piece of software for navigating a collection composed of multiple family photo collections using a customisable family tree as an input. This project is an investigation of how the iterative design model impacts requirements, usability and software design over the iterations. The software will be tested with users at each iteration, reflecting heavily on how the aforementioned attributes are impacted by the development cycle.

Acknowledgements

I would like to thank Maria Wolters for her supervision on both this project and the Undergraduate Research Practical in 2014. I thank my family, friends and my colleagues at The New England Historic Genealogical Society for their time and effort in user testing the application. I would finally also like to give thanks to Chelsea Van Baush and Lesley Inglis.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Michael Inglis*)

Table of Contents

Chapter 1

Introduction

Designing and implementing an application which deals with the abstract of this project has to take significant factors into account. I aim to present an in-depth look at the design, implementation, user testing and the re-defining of requirements after each of the iterations which will take place within the project's lifecycle.

Developing this application using an iterative method will steer it towards a more intuitive, useful piece of software which is built from user feedback. This will allow the application to be created with the user in mind, something which cannot be guaranteed with the waterfall model.

I will look at the pros and cons of each software development process in Chapter 2 before moving onto my reasoning in selecting an iterative design cycle. I will also spend Chapter 8 looking at how the iterative design method influenced the project overall. In chapters 3 through 7 I will outline my requirements, design, implementation and user testing methodologies before investigating the effects of the iterative development cycle on the software design, usability and formation of the next iteration's set of requirements.

This piece of software builds its foundations on an Undergraduate Research Practical (URP) titled "Managing and Preserving Personal Photo Collections for Future Generations" [1], written by myself in 2014. While the paper predominantly looks at how users organise and view their photo collections. I will focus mainly on the high level specification for a software application shown at the end of the paper, deriving the requirements to suit the platform of choice: iOS.

I was fortunate enough to also perform user testing with The New England Historic Genealogical Society (NEHGS) for the first two iterations. Given their vast experience in using and developing software related to genealogy, their unique feedback on the family tree interface was extremely useful in tailoring the user interface to be as intuitive as possible.

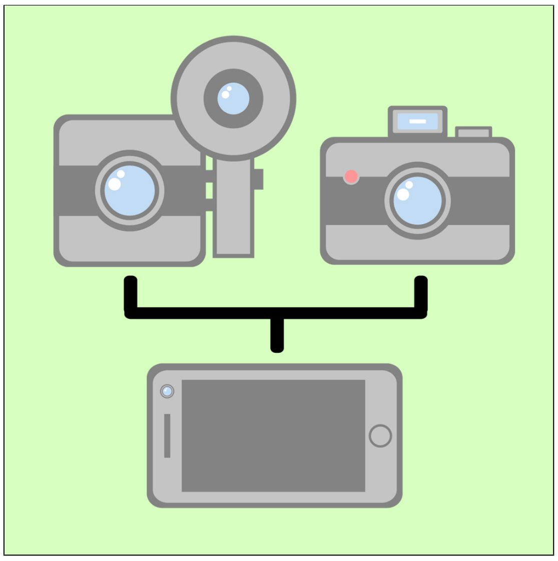


Figure 1.1: Application logo: Photo moments across a family tree

Chapter 2

Rooted in Iterations

2.1 Introduction

In this chapter I will outline the research and reasoning in choosing an iterative design model over the more traditional Waterfall model of software design.

I will look at how both processes compare to each other at each step of software development before expanding onto my reasoning for picking the iterative model with respect to my software development intentions. The Spiral Model was not considered as it tends to suit projects with a high cost and high susceptibility to change, something this project did not have.

The paper I will be looking at and citing throughout this chapter is titled "A Comparison Between Three SDLC Models Waterfall Model, Spiral Model, and Incremental/Iterative Model"[2] and was written by Adel Alshamrani and Abdullah Bahattab.

2.2 Methodology

2.2.1 Waterfall

The author regards "the special feature of this model is its sequential steps". He states that "it goes downward through the phases of requirements analysis, design, coding, testing, and maintenance". Contrasting what we will see below is that "stages that construct this model are not overlapping stages, which means that the waterfall model begins and ends one stage before starting the next one".

2.2.2 Iterative

The paper goes onto suggest that the iterative model "combines elements of the waterfall model in an iterative fashion", stating that the model "constructs a partial implementation of a total system. Then, it slowly adds increased functionality". The process of this model of development is then declared as "each subsequent release will add a function to the previous one until all designed functionalities are implemented".

I will explore the elements of each model, contrasting the waterfall methodologies against the iterative ones before concluding at the end why the iterative model was chosen to be explored for the development of this project. Maintenance has been excluded due to its irrelevance to this project.

2.3 Requirements

The definition of the requirements is stated as "a description of a system behavior to be developed".

2.3.1 Waterfall

In the waterfall model, the author writes that "all requirements must be known upfront." This results in the project being inflexible after the requirements stage is completed since stages cannot be overlapped or repeated. This was something this project would not have been able to cope with given that it was unknown whether or not the new functionalities were initially viable at each requirements stage.

2.3.2 Iterative

Iterative development "requires early definition of a complete and fully functional system to allow for the definition of increments." This is given within the Undergraduate Research Practical (URP)[1]. What is not available however is a format in which the requirements can be conveyed to suit the iterative development. This adaption is something that is looked at in the subsequent chapter.

Unlike the waterfall model, it is possible to return to the requirements stage after every iteration. This leaves the project much more flexible to change using the feedback at the user testing stage or any problems which may arise in the design and implementation stages.

2.4 Design

The paper's definition of design is "the gathered information from the previous phase is evaluated and a proper implementation is formulated. It is the process of planning and problem solving for a software solution."

In my iterations in the following chapters, I will explain much of the Software Design methodology based on this quote. This is the step of solving the conceptual problems that arise in translating my requirements into something that can be implemented.

2.4.1 Waterfall

In the waterfall methodology, it is stated "define before design, and design before coding". He goes onto suggest however that "it is not a preferred model for complex and object-oriented projects". Both of these points prove problematic with this project. First of all, with respect to usability, how could something be refined to suit the user if it is simply explained to them at the requirements gathering stage over observed usability testing and questionnaires? Secondly, the design of this project, based on the initial requirements, will result in complex, object-oriented software.

2.4.2 Iterative

Aside from being open to handling the above counter arguments regarding usability and object-oriented design, the iterative model must "develop high-risk or major functions first". This allows not only the most robust features of the software design to be the main ones as they are finished first and thus tested the most, but ensures the main project functionality is finished by prioritising it over and above everything else.

2.5 Implementation

The Implementation is simply using the Design as a blueprint to create the requirements in code. Much of my implementation sections will involve the methodology in which the design was developed into code. I will also consider problems that occurred beyond the design, reasoning and evaluating design decisions with respect to the code and the user interface.

2.5.1 Waterfall

Both software design methodologies here follow similar points at this stage. The waterfall model however is stated as "being a linear model, it is very simple to implement". Implementation only occurs once within this model, between the Design and User Testing stage. This requires for all of the features to be implemented and functioning as required, something that in theory - assuming the prior stages have been carried out correctly - should be straightforward, but in practice is often difficult as "small changes or errors that arise in the completed software may cause a lot of problems".

2.5.2 Iterative

Implementation however occurs as many times as iterations in iterative development. While the paper states it "requires good planning and design", this can be far less thorough than the waterfall method given the consequences of a bad implementation are not necessarily detrimental to the project as a whole.

In iterative development, the author states “risk is spread across smaller increments instead of concentrating in one large development”. This suits both the style of the project, focusing on the main parts of the application while also taking into consideration outside factors such as time to develop.

2.6 User Testing

The definition of the user testing phase in the paper is that it “deals with the real testing and checking of the software solutions that have been developed to meet the original requirements”. With respect to my own testing in each iteration, I will do just this; evaluating the feedback of the users against my own set of requirements.

2.6.1 Waterfall

Due to the problems that were outlined in the implementation section with regards to the waterfall model, it is understandable that “customers may have little opportunity to preview the system until it may be too late”. This feeds into the argument of whether or not the main functionality was prioritised and if, after implementing the requirements, the software performs as expected. This is explained in the paper, where “backing up to solve mistakes is difficult, once an application is in the testing stage, it is very difficult to go back and change something that was not well-thought out in the concept stage.”

2.6.2 Iterative

The paper here suggests that “each release delivers an operational product” and “lessons learned at the end of each incremental delivery can result in positive revisions for the next increment”. This enables the feedback at the user testing stage to be used to refine the project’s functionality and usability, giving the chance to redefine requirements in the early stages as to avoid problems later on in the development.

2.7 Conclusions

While overlooking other elements such as cost and timing to market due to irrelevance to this project, it should be noted that the guarantee of success is high with respect to deployment on iterative projects and low on waterfall projects.

This due to many of the factors explored above but comes down to predominantly risk. While in the waterfall method, the "requirements are very well known, clear, and fixed", this is only beneficial "when quality is more important than cost or schedule". Taking into account however time and cost results in the author remarking that it is "risky to develop the whole system at once".

The iterative model however bypasses this aspect of risk with respect it time, cost and core functionality. It "reduces the risk of failure and changing the requirements." In describing the iterative process, the paper alludes that "each linear sequence produces deliverable increments of the software. The basic requirements are addressed in the first increment, and it is the core product, however, many supplementary features (some known, others unknown) remain undeliverable at this increment".

This is how I aim to structure this project, attempting to address the main requirements in the First Iteration before moving onto further, peripheral requirements in the subsequent iterations.

I will also structure each iteration's chapter in a similar format as seen above, looking at, and evaluating each of the individual steps and investigating the effect of the iterative development on the software's requirements, design and usability.

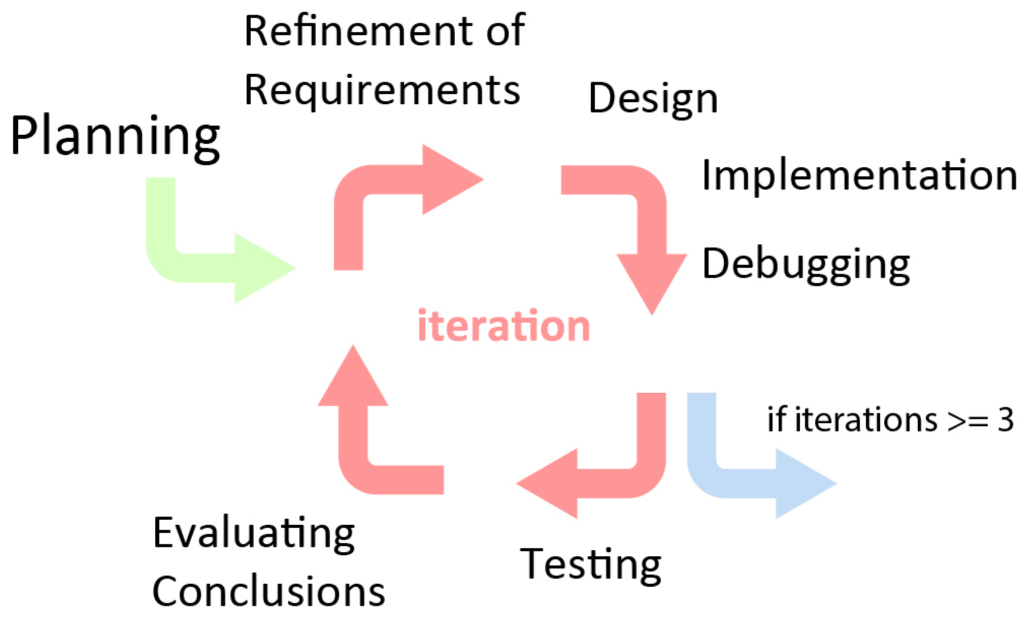


Figure 2.1: The project's iterative development cycle

Chapter 3

Iteration 0: Planting Photo Collections for Future Generations

3.1 Introduction

In this chapter I will focus on my reasoning and methodology in translating the high level software specification seen in the Undergraduate Research Practical (URP)[1] into a viable set of functional requirements to be implemented over the duration of this project. I will take into account factors such as time and scope in selecting functionalities that are suitable to be implemented using the iterative software development model.

3.2 Platform

As noted throughout the findings within the chapter titled "How People Store and Manage their Photos" within the URP, we see that very few people will actively store and manage their own photo collections. This was the primary reason Apple's framework was considered to build the application on.

It firstly organises all the photos automatically based on metadata into what are called "Moments". The documentation associated with Apple's Photos Framework[3] contains a definition for “Moments” which is as follows:

“The Photos app automatically creates moments to group assets by time and location, and also creates moment lists to group related moments. Moment lists have two subtypes: a moment cluster groups a few related moments, and a moment year groups all moments in a calendar year.”

What we will call and refer to as “Moments” for the rest of this project however is simply the grouped photos by date and place.

Apple’s framework also spans across both the desktop and mobile platforms. This is useful because mobile devices are the source of most of the photos and desktop devices are the main source of storage as seen in the same chapter of the URP mentioned above.

An iPad Air running iOS 9.0 was thus selected as the platform for the application. This also allows for the complete scrapping of the URP’s “Event” organisational structure in favour of Moments. We can now begin to specify details which were previously based on assumptions within the URP.

3.3 Deriving Requirements

3.3.1 Problems to Consider

3.3.1.1 Importing Images

Due to how iOS handles image imports, we are unable to drop photos into the application as stated in the URP and instead have to rely on the user syncing their images to the iPad using iTunes. Any alteration to the photos as regards to metadata must be carried out on the synced computer as the iOS device will simply duplicate the photo with the newly changed attribute and re-add it to the collection. We are also unable to delete photos which have originated from the syncing to the device.

These restrictions within the platform cause us to pivot our functional requirements and instead develop an application which will be used primarily for viewing and organising photo collections, not for the input, dating and geotagging of photos; tasks Apple's desktop application "Photos" - released after my URP - can efficiently manage.

3.3.1.2 Tagging Faces

One precondition on the photo set which is vital to how this application will function is that the added family members' faces must be tagged within the photos. This could be done in one of 3 ways and all were extensively considered;

1. Firstly - and most convenient to us - is that the user tags the faces within the Photos OSX application using apples "Faces" system. This creates an album on the synced iOS device which can be accessed using the Photos Framework. This is the closest to the methodology outlined in the URP with respect to Moments over the Event structure.

2. The second option was adding an automatic face tagger. This would involve a pop up interface which would list faces automatically, pulled up by face detection tools within Apple's Photos Framework. This would pop up occasionally and can be dismissed immediately in an attempt to be as unobtrusive as possible. We would have to however find a way of enticing users to use such a feature given the importance to application functionality.

3. The third and final method of adding faces is to compensate for the fact the auto face detection won't have 100% accuracy. Users would be able to go into individual photos and add tagged faces.

While a combination of the three was at one point considered, for the sake of this project, it was decided that the first option was the most beneficial to us given that the functionalities listed in options 2 and 3 already existed in Apple's Photos application - the software users would have to do option 1 on.

This allows for the time I have available to be better spent on exploring new concepts such as the search and display of photos all while benefiting from software which does what it does better than I could ever do with the time available.

3.3.2 Refining Requirements

This change in requirements from an application which organises photos based on how they are imported to an application which is used to view already imported photos - but in the same organisational structure as we previously wanted - actually benefits the conclusions found within the URP as regards to how users manage their photos:

"there was two polarising users in terms of how photos are managed; one meticulously organised and the other didn't have any form of organisation".

Cutting out the need for photos to be manually imported into specific events bypasses much of the manual work we would have required users to do in the URP specification. Instead, we can rely on the "Moments" within Apple's Photos Framework to automatically organise the photos with respect to their metadata. The only manual work the user now has to carry out with respect to organsing their photos is the tagging of people's faces within the OS X Photos application, something Apple handles very elegantly.

Our unique organisational inputs and outputs however remain the same as the URP; a family tree is inputted and the timeline associated with that family tree can be displayed. This forms the basis of what I will develop and can be seen in Figures 3.1 and 3.2 respectively.

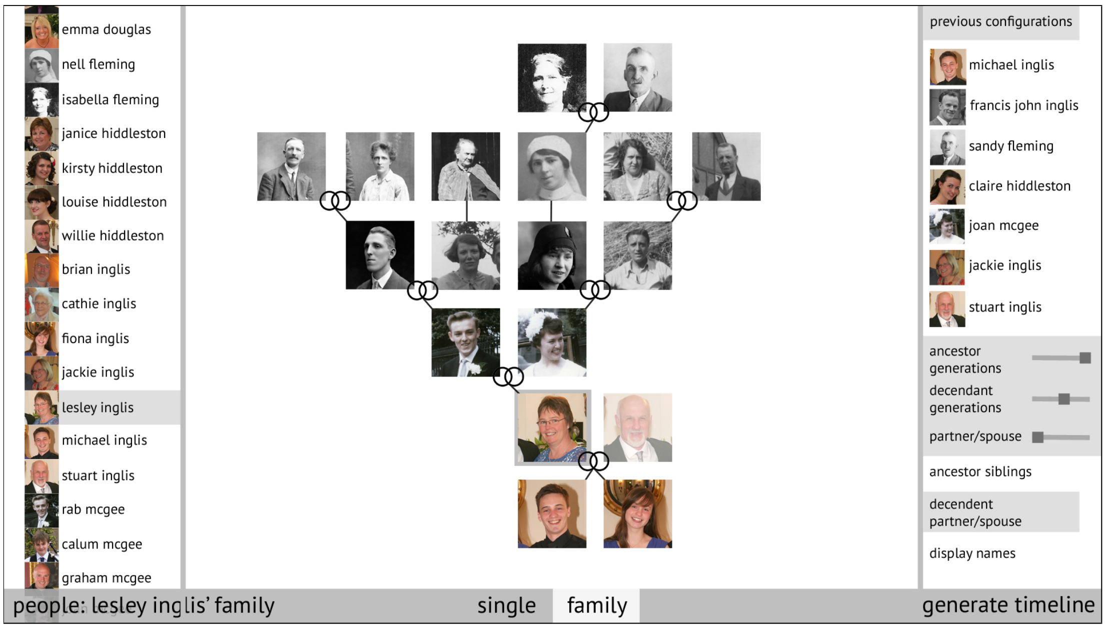


Figure 3.1: URP - Display Family

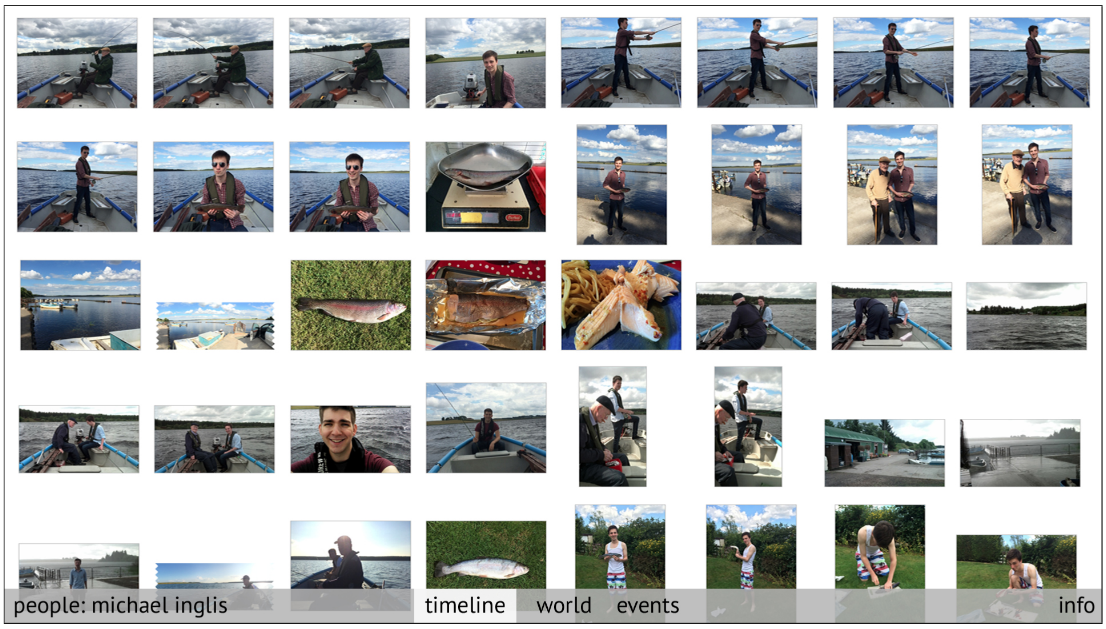


Figure 3.2: URP - Collection Display Timeline

In an effort to have a quick turnaround for user testing and in anticipation of further iterations within the iterative design cycle, the features of the application were split up into two distinctive modules; the Family Tree Module and the Photo Display Module, both taking inspiration from two images above. The Family Tree Module will serve as the whole application for the first and second iterations.

The Photo Display Module will be included within the third iteration due to the fact the Family Tree Module is required to be finished before it is functional.

What I am proposing is functionally identical to what we see within the Photos application on iOS or OSX with the exception that it takes the family tree of people as an input to dynamically generate the timeline based on the family members contained within.

3.3.2.1 Family Tree Module

The functional requirements of the Family Tree module are largely abstracted over within the specification of the URP. What it does state in Use Case 2 however is as follows:

“If the “family” option is selected over the default ̈singleöption, a family tree interface for generating a Collection Display is brought up. Users can select a person and customise the family tree which is associated with that person, tweaking elements such as number of generations of ancestors/descendants and the level to which their partner/spouse's Events are included."

This methodology in question can be easily visualised using the associated concept render from the URP in Figure 3.1. Elements of this design as regards to the tree itself and the list of people on the left from which the family trees can be generated from will serve as the starting point for the development of the family tree application.

Functionalities such as adding people or assigning relations were not mentioned in the URP and must be considered within the first iteration. Aspects such as associating people with their respective album of tagged faces will also have to be considered now that the photo organisational component will require as such.

3.3.2.2 Photo Display Module

This module takes the input of an instance of a family tree, passed from the Family Tree Module and creates a timeline based on the Persons within the family tree.

The unique methodology in which timelines will be drawn remains similar to what is outlined within the URP, involving Moments however over the proposed Event structure.

The photo timeline doesn't take into account each individual photo the Person is tagged in, but the whole Moment associated with the photo. This is the best method of giving us more of a "photo collection". It gives us photos which also contain no people over simply returning a collection of tagged faces, something Apple's Photos application already does.

3.3.2.3 Non Functional Requirements: Usability

One of the major factors that will form the most consistent Non-Functional requirement throughout the lifecycle of the project is usability.

Consistency and minimalistic design will form the basis of the motivation behind the user interface. One of the main elements of human-computer interaction I wished to explore is learnability. This is due to the requirement of inputing a family tree before any of the application's real functionality can be explored, requiring a complete system to be learned before any user satisfaction can occur.

The ISO 25010[4] definition of learnability is the "degree to which a product or system can be used by specified users to achieve specified goals of learning to use the product or system with effectiveness, efficiency, freedom from risk and satisfaction in a specified context of use". These elements of usability, effectiveness, efficiency and satisfaction must all be considered if the interface is to be successful.

The element of usability this project will considering a priority is efficiency. Gilbert Cockton states in The Encyclopedia of Human-Computer Interaction that "As regards a more realistic extent of usability (as opposed to binary usable vs. unusable), we must trade off factors such as efficiency, effectiveness and satisfaction against each other". This trade off is something I will consider extensively in order in to increase efficiency and learnability through a minimalistic design. This will require effectiveness and satisfaction to be reduced somewhat in how information is displayed and manipulated, creating a UI - especially with respect to the family tree module - which, while extremely efficient in order to get onto the photo functionalities, could be perceived as boring.

3.4 Conclusions

The basic set of requirements listed above combine to what I've called "Momentree" and can be seen in Figure 3.3 below. This includes an example family tree with two people, each tagged in three photos, all of which are then associated with three Moments.

The application takes its name after the algorithm I developed to carry this out in Iteration 3. This functionality, being able to display a photo collection using a family tree is what separates this application out as unique from other photo viewing softwares that exist.

This functionality however is not possible without a robust family tree builder, the development of which will form the basis for the next two chapters and can be seen included with the photos functionality in the use case in Figure 3.4.

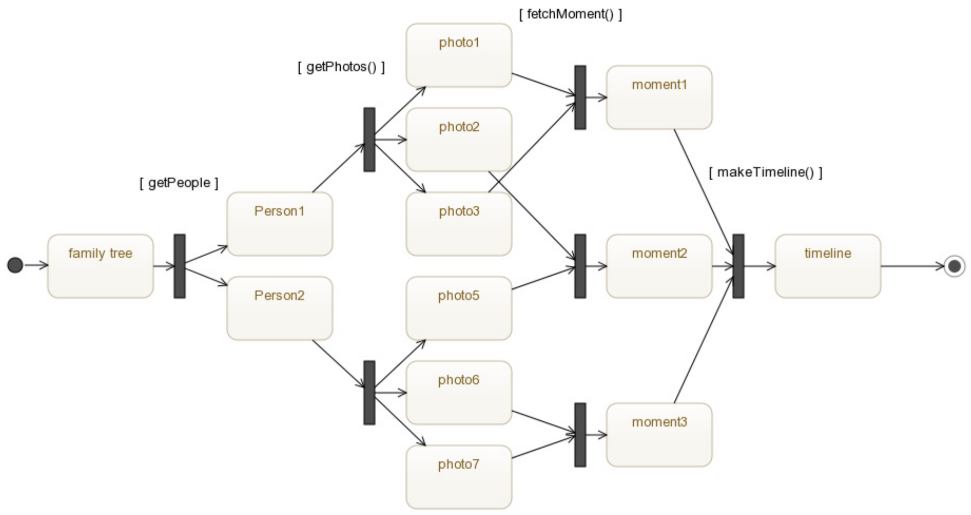


Figure 3.3: Diagram of “Momentree”

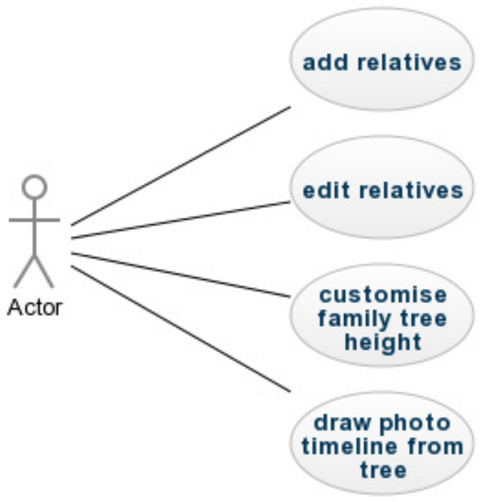


Figure 3.4: “Momentree” use case

Chapter 4

Iteration 1: Growing Family Trees

4.1 Introduction

In this chapter I look at the design and implementation challenges I faced in developing my application for the first round of user testing with The New England Historic Genealogical Society (NEHGS).

4.2 Requirements

The requirements for this Iteration are derived from the previous chapter which were in turn formulated by looking at turning the Undergraduate Research Practical (URP)[1] into a viable application for iOS.

What was described as the “Family Tree Module” will form the foundation of this chapter and the next. In an effort to have a functioning testable application before the second iteration, the Functional and Non-Functional requirements were scaled back to the bare minimum, concentrating only on functionality over any expanded features.

4.2.1 Functional Requirements

- A basic family tree builder which allows users to add a person, edit a person and assign relationships between persons.
- A diagram on which the relationships can be visualised as a family tree.
- A list of all persons whereby clicking one dynamically changes the selected person to the root of family tree diagram.

4.2.2 Non Functional Requirements

- A high degree of usability with a focus on quick learnability.
- Minimalistic in design, focusing only on the required features.

With this set of requirements, it was possible to start the first iteration of the development cycle by starting with the software architecture design.

4.3 Design

The software design of the system was conceived completely from scratch, ignoring all of the suggested design in the Undergraduate Research Practical (URP)[1] due to the move away from the Event hierarchy. The structure of the software was determined using the Model-View-Controller (MVC) design pattern (see Figure 4.1). In an effort to best describe the architecture I will document each part of this in depth.

4.3.1 Model

The Model comprises of two classes, Person and FamilyTree which interact with each other as seen in Figure 4.2.

Person holds all the required information about a specific person's immediate family. Persons are assigned to a Person by creating a new relation (father, mother, partner). This relation can also be removed and will break the association between the two Person objects.

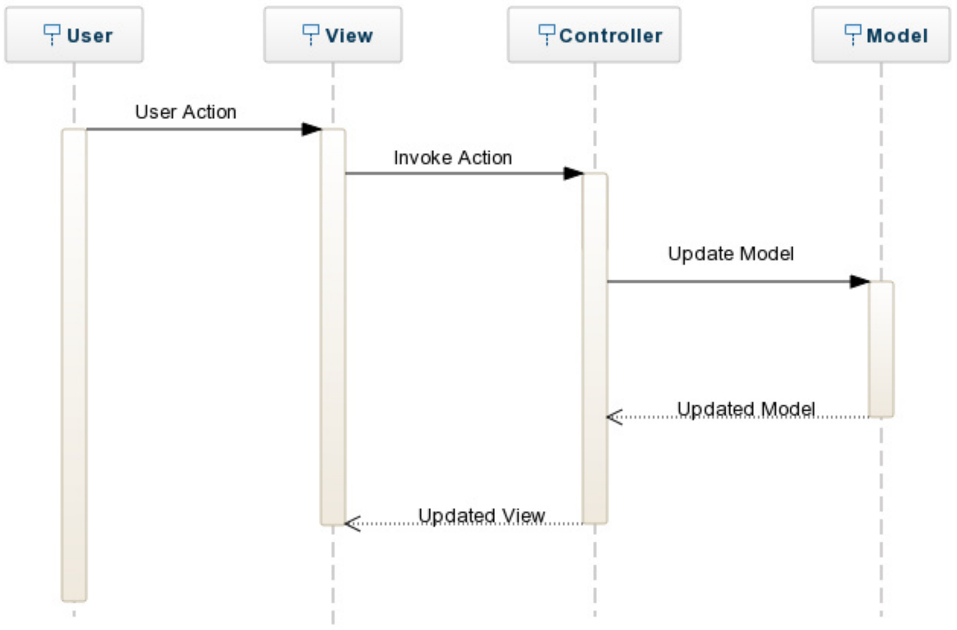


Figure 4.1: MVC Sequence Diagram of Software

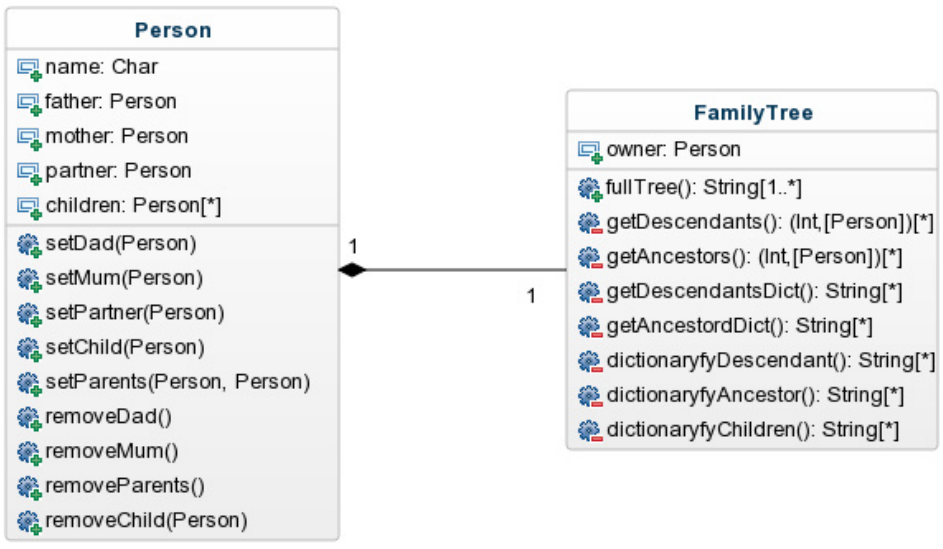

Person
name: Char
father: Person
mother: Person
partner: Person
children: Person[*]
setDad(Person)
setMum(Person)
setPartner(Person)
setChild(Person)
setParents(Person, Person)
removeDad()
removeMum()
removeParents()
removeChild(Person)
1
1
FamilyTree
owner: Person
fullTree(): String[1..*]
getDescendants(): (Int,[Person])[*]
getAncestors(): (Int,[Person])[*]
getDescendantsDict(): String[*]
getAncestordDict(): String[*]
dictionaryfyDescendant(): String[*]
dictionaryfyAncestor(): String[*]
dictionaryfyChildren(): String[*]

Figure 4.2: Iteration 1: Person & Family Class Diagrams

The Family Tree Class takes input of a Person and generates a FamilyTree based on that person. The class has two important methods, getAncestorsDict() and getDescendantsDict(), both of which are functionally similar, calling getAncestors() and getDescendants() (see Listing 4.1) which carry out Breadth First

Search on Person.mum/Person.dad and Person.children respectively.

This generates a list of names with their generation depth as a tuple. These lists can then be passed to the "dictionaryfy" methods which place their input into dictionaries, using the relations as keys. Both of the private "Dict" methods are called within the fullTree() method which combines them into one large ordered array of dictionaries in anticipation of them being converted to JSON for the family tree view.

Listing 4.1: Breadth First Search

```
func getDecendants() -> ([(Int,[Person])]) {
    var iteration = 0
    var thisLevel = [Person]()

    var decendantListWithGeneration = [(Int,[Person])]()

    thisLevel.append(self)

    while !thisLevel.isEmpty {
        var nextLevel = [Person]()
        for n in thisLevel {
            for c in n.children {
                nextLevel.append(c)
            }
        }
        decendantListWithGeneration.append((iteration, thisLevel))
        thisLevel = nextLevel

        iteration += 1
    }
    return (decendantListWithGeneration)
}
```

The Breadth First Search (BFS) was selected for two reasons. Firstly, it allows us to traverse the tree in such a way that the structure of the generations can remain intact - a feature which will be looked at in Iteration 2. Secondly, BFS has time complexity of O(n)[5] on a binary tree with nodes n. This speed is the same as Depth First Search (DFS)[6], but DFS would not allow the structure of the generations to be preserved. I did however consider that given the node set was so small, it was unlikely that any search algorithm would be detrimental to the software's usability.

4.3.2 View

The view comprises of three screens, the Family View, Family Builder and Person Editor each of which is required to implement an intuitive family tree builder in its most basic form. The states of the application with respect to screens and their respective transitions can be seen in Figure 4.3.

In deriving the requirements for this first iteration, concept renders were created for each individual screen. This gave focus to the user interface for the first time since the renders were created in the URP and allowed me to properly consider what functionalities would be usable within the first round of user testing. I will look at in length how each screen evolved into its final design (as seen in application screenshots) in the Implementation section of this chapter.

The FamilyView screen consists of what we see in the concept render in Figure 4.4. I wanted a list of inputted Person objects on the left and the option to add more at the top of that list, bringing the user to the FamilyBuilder screen. The bulk of the FamilyView screen was also intended to be taken up by the FamilyTreeView which would draw the full family tree of whatever person was selected on the list.

The design adheres closely to the iOS standard. With the list of people on the left and the main content (in this case the FamilyTreeView) on the right. The tree dynamically changes based on the person selected on the left. The "+" used at the top of the Person list is iOS standard for an addition to the list. Selecting this leads to the FamilyBuilder screen.

The FamilyBuilder screen deals with the addition of a Person to the list. Prioritising functionality to have a testable build at this stage led to the Person's name and parents being the only attributes that are intended as Person.children would have to compensate for an unspecified amount of children.

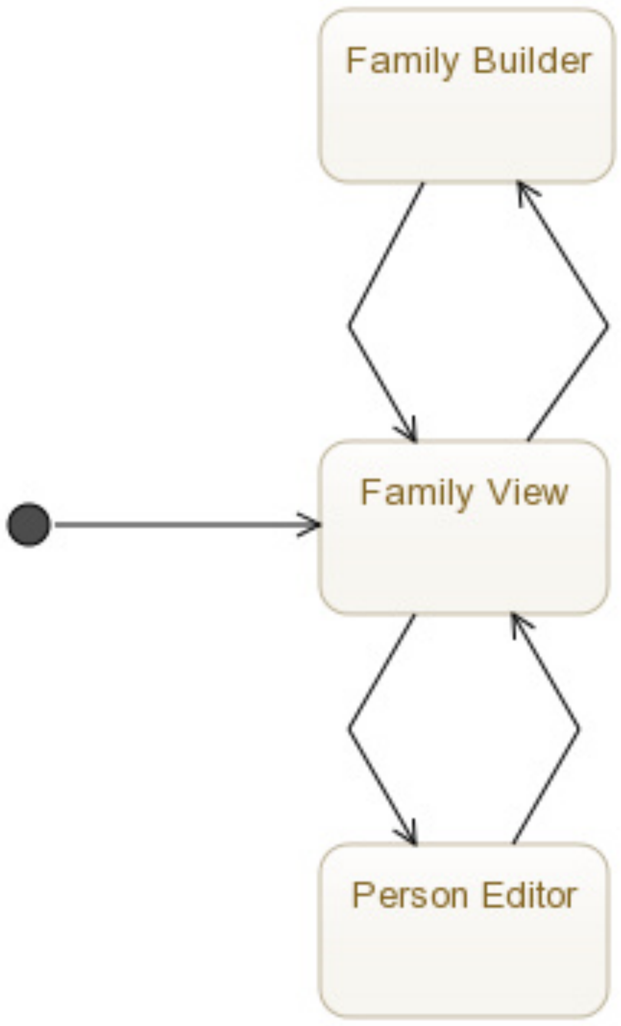


Figure 4.3: Iteration 1: View Transitions

Moving back to the FamilyView screen having selected ‘save person’ on the top left, we can see the small ‘>’ leads to the PersonEditor screen. This is much like the FamilyBuilder screen, with the exception of the name input, simply allowing the user to change the Person’s parents. Selecting ‘save person’ would then bring us back to the FamilyView screen.

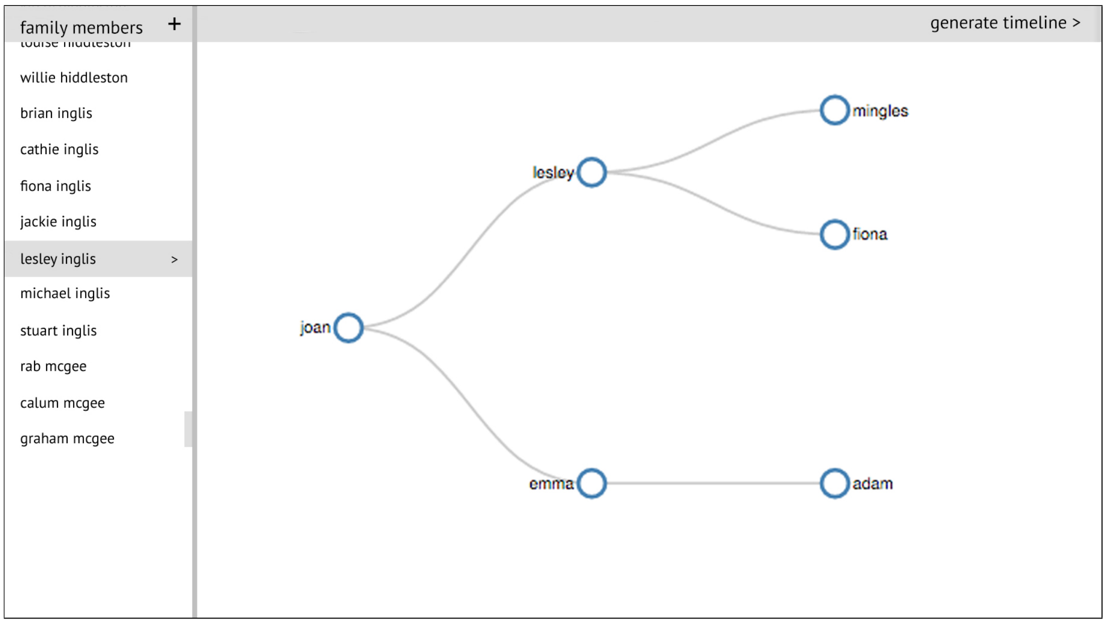


Figure 4.4: Iteration 1: Concept Render FamilyView

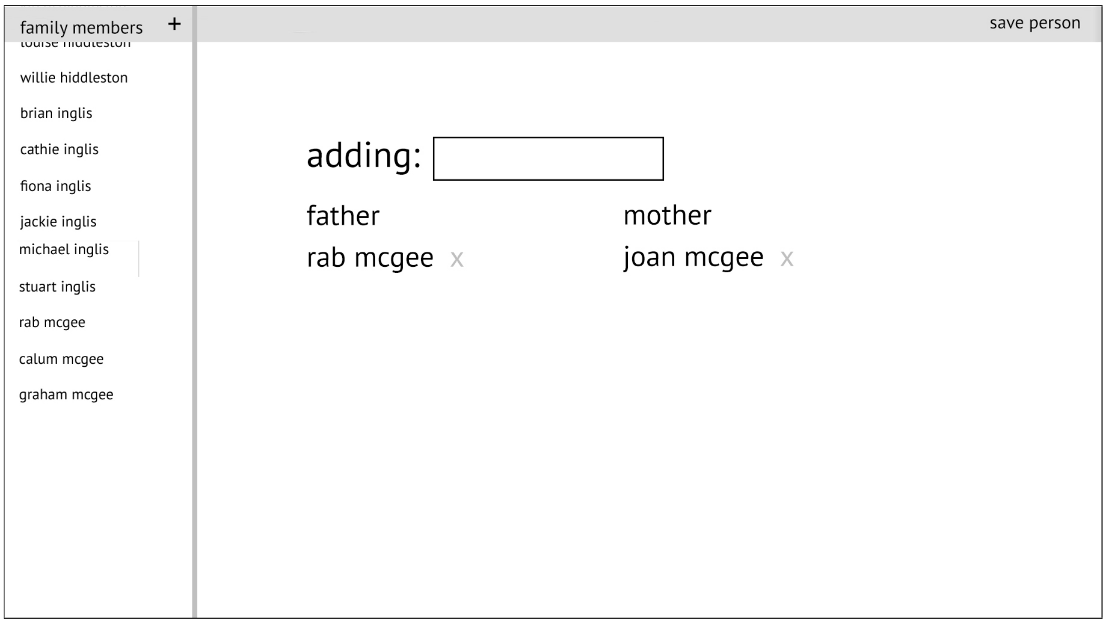


Figure 4.5: Iteration 1: Concept Render FamilyBuilder

4.3.3 Controller

Each respective view has its own controller, controlling the actions between the view and the model.

The FamilyViewController is required to do only two things; organise the data from the models and then present it to the view. The controller is passed a

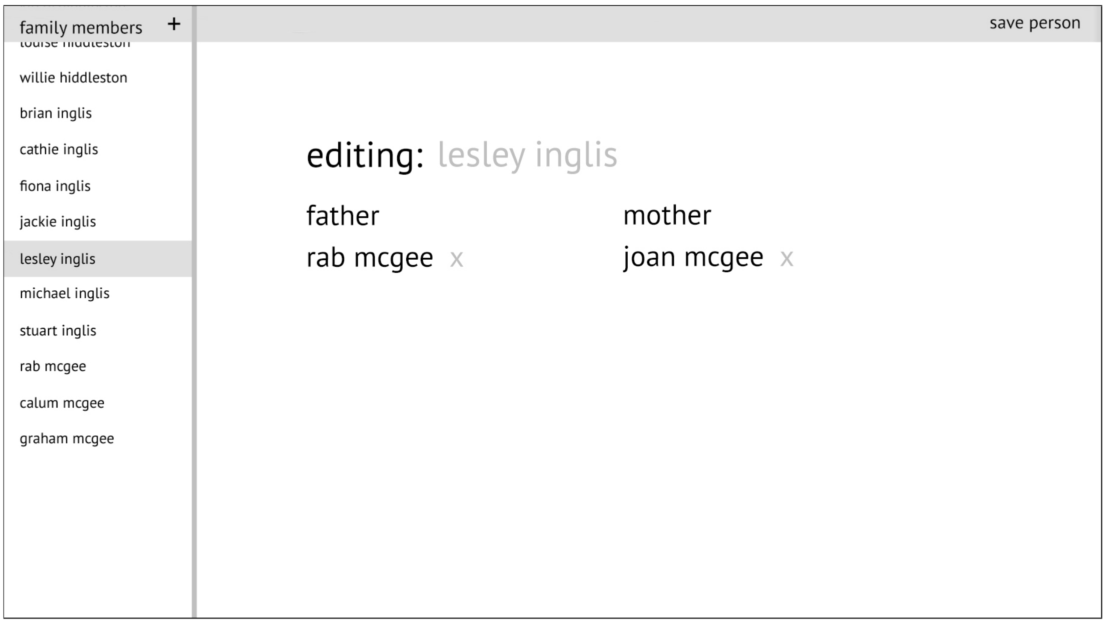


Figure 4.6: Iteration 1: Concept Render PersonEditor

'selected Person' from the views via user input. This is used to create an object of class FamilyTree, from which the full family tree is derived as a dictionary and presented in JSON. This is then loaded into the 'Graph Drawer' which will be looked it in the Implementation section of the chapter.

Both the FamilyBuilderController and the PersonEditorController do as we expect given the similar functionalities on their views.

The FamilyBuilderController simply saves the name, mother and father inputs from the user on the FamilyBuilder screen, creates an object of Person and appends it to the list of people displayed.

The PersonEditorController presents the mother and father attributes of the person and updates them to whatever change has taken place when the screen is returned to the FamilyView screen.

4.4 Implementation

The aforementioned classes, Person and FamilyTree were implemented in Swift according to the design seen in the class diagrams. Both were originally implemented in Python in an attempt to ease me into developing code in Swift, a language I was unfamiliar with before this project.

A significant test suite was also developed before any Swift was written. Focusing on Behavioural Driven Development, a habit of collaborative projects. I would create a new test demonstrating a specific behavior, run it expecting it to fail, write the minimal amount of code to make the test pass and run it again, expecting the new test to pass along with all the other tests.

This methodology was implemented heavily on the tree generation, ensuring the JSON passed to the controller was exactly what would be expected given the input. While this increased development time in the beginning, it resulted in quicker implementation of new features as the whole software project didn't need to be bug fixed and any problems were isolated to the new features.

Attention was also given to the version control, committing to Github after each successful implementation of a feature. This ensured I always had a working backup in case of system failure. While this is crucial for collaborative projects, I believe it is good practice to utilise it fully on personal projects.

As regards to the view, the whole design was implemented within an iOS storyboard. Each scene is connected to each other using the implemented controllers described in the Design section.

The FamilyView screen (see Figure 4.7) was arguably the most complicated part of the view/controller implementation. As well as linking both the FamilyBuilder and PersonEditor screens via the "+" and "Edit Person" buttons, this screen handles the complete display of the global attributes PersonArray(an array of all of the inputted Person objects) on the left and from that, the SelectedPerson's family tree on the right.

The list of Persons on the left was implemented using a table view which iterated through the PersonArray, creating a row object of a custom class PersonCell which specified the name. The PersonCell class was made in preparation of potentially having photos to the left of the name in later iterations. This was why it was picked over using a pre-existing row element.

Developing the FamilyTreeView on the right of the FamilyView screen and linking it to the FamilyTree class' outputted JSON was without a doubt the most complicated element of the project at this stage.

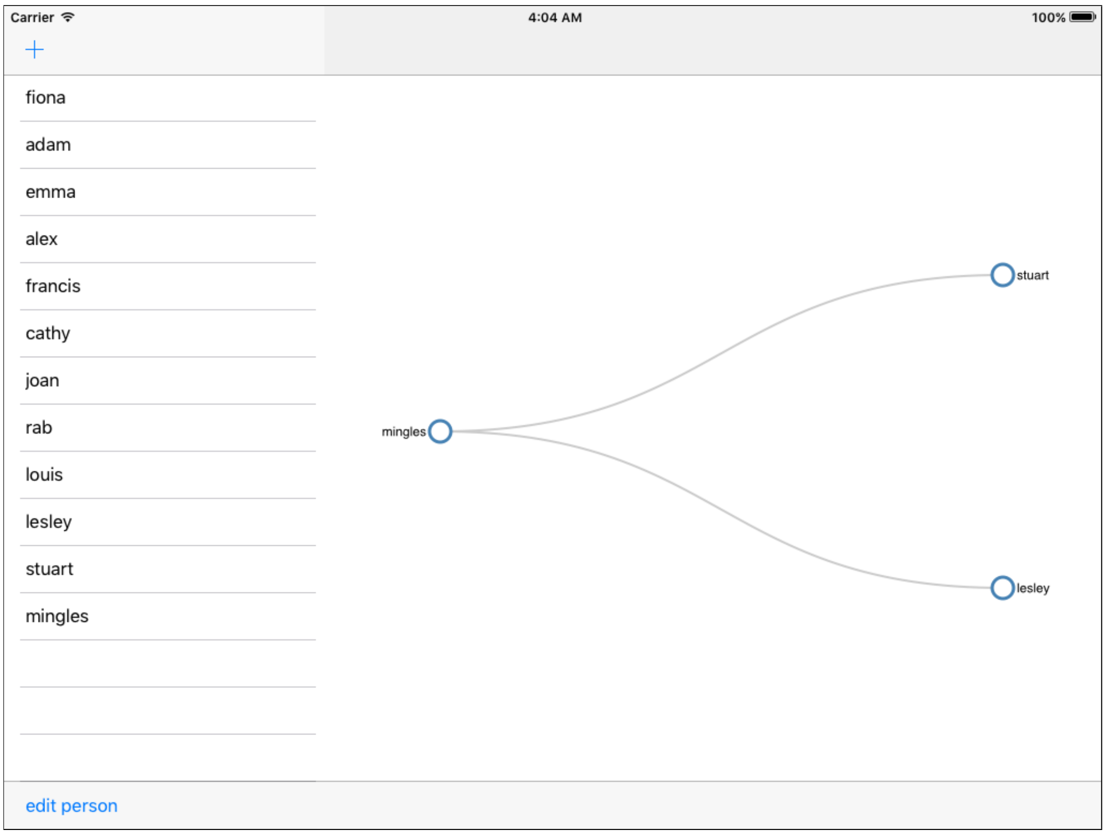


Figure 4.7: Iteration 1: FamilyView Screenshot

My initial approach was to attempt a graph drawing algorithm within Swift myself, calculating the size of the graph required on screen before drawing it in a grid. This however became untenable due to my inexperience with the new language and a lack of Swift libraries due to it only having existing for a couple of years. I instead opted for a local HTML file which is loaded into a webview each time the FamilyViewController class is invoked. Within this HTML file, I was then able to load in D3.js[7] - a Javascript visualisation library for drawing graphs - and use it to parse the JSON string and output a single binary tree with the associated names.

A problem with this implementation of the tree however was it was only capable of displaying either ancestors or descendants at once. This could result in a full tree, but only if the given person was the highest generation ancestor or lowest generation descendant depending on which way the tree was set up. Due to time constraints and for the sake of user testing, when a participant inputed their own tree, I opted for drawing only descendants and the root of the tree to be their highest ancestor generation wise of the selected person. This small cheat resulted in all the participants being able to see their own family tree even though the software was actually drawing the tree for what was probably their grandfather.

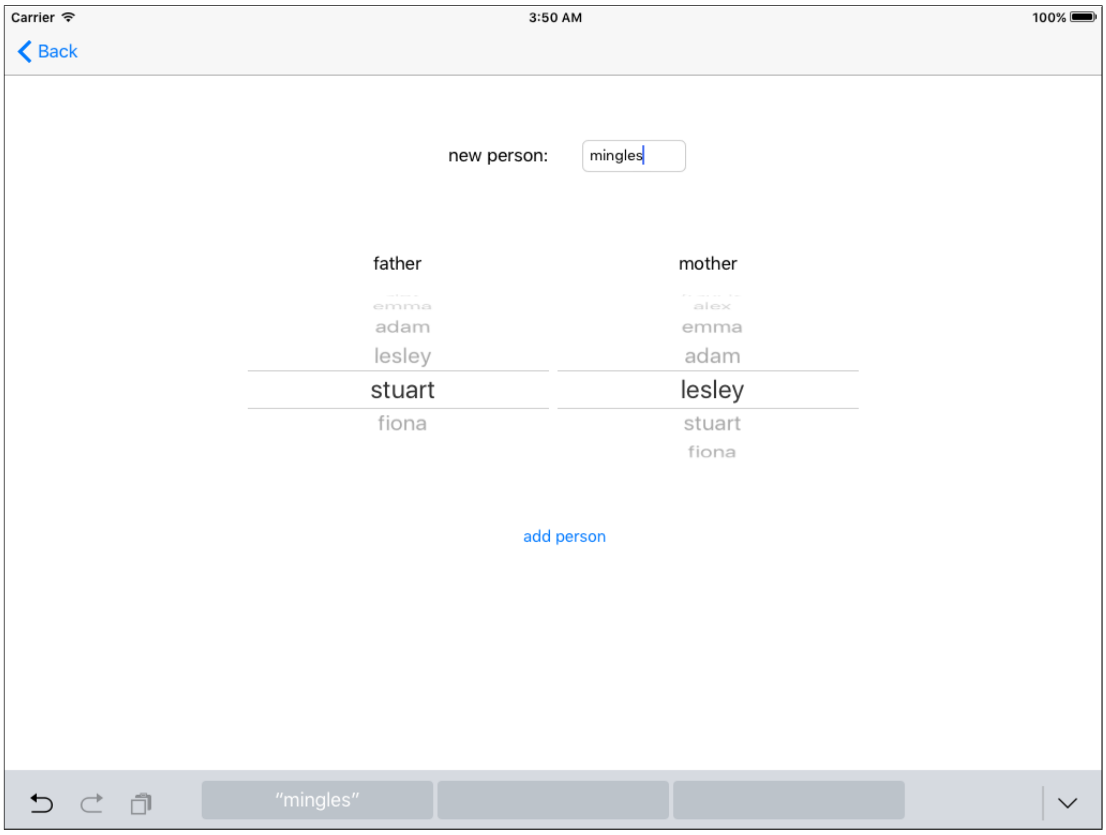


Figure 4.8: Iteration 1: FamilyBuilder Screenshot

Most elements used within the FamilyBuilder (see Figure 4.8) and PersonEditor (see Figure 4.9) views were very straight forward, implementing buttons and text boxes as well as the scrolling ‘picker‘. The picker was selected as an interesting way to display a list, utilising a more interactive touch screen gesture than simply typing in the Person’s name into an autocorrect box, something I tried to avoid given the fact a touch screen keyboard takes up half of the screen.

4.5 User Testing

4.5.1 Background

Given the nature of the software in the current state, it was in essence simply a basic implementation of a family tree application. I was able to use my prior

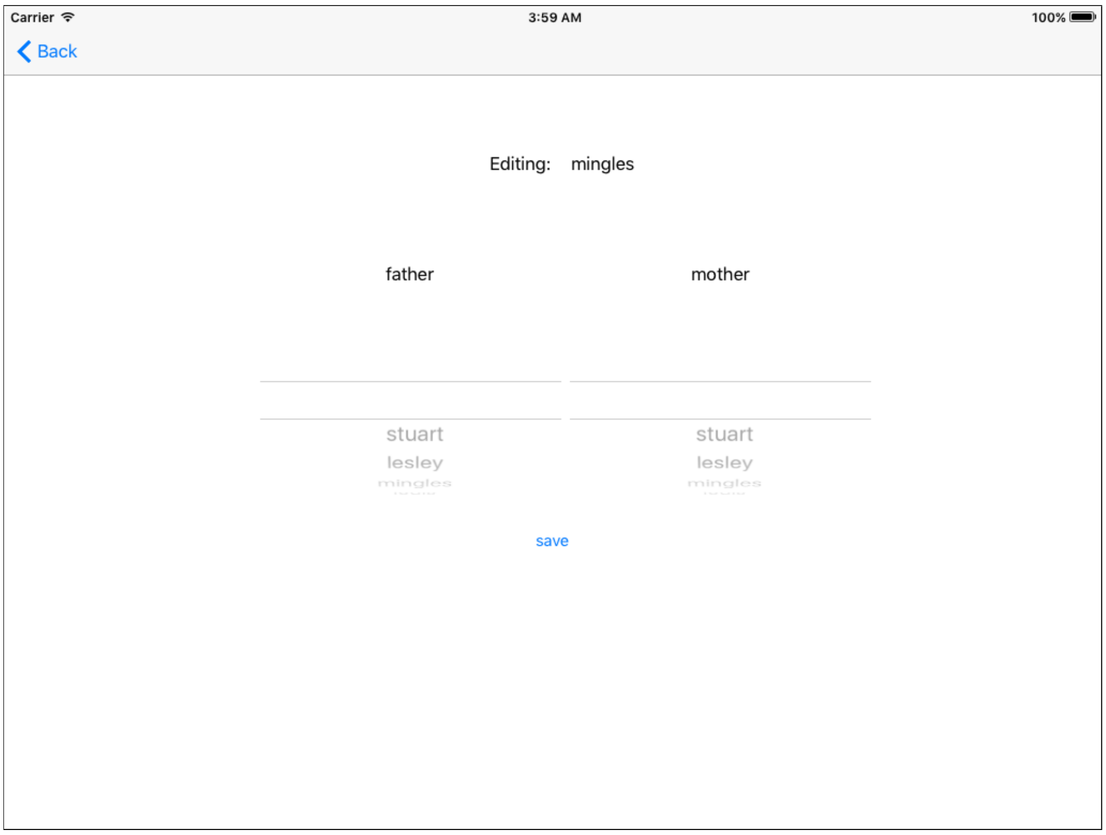


Figure 4.9: Iteration 1: PersonEditor Screenshot

work experience with The New England Historic Genealogical Society (NEHGS) in Boston, Massachusetts to organise and carry out user testing sessions with five members of their software development team, who are responsible for creating and maintaining their online genealogical website, American Ancestors[8] . The experience my participants had as regards to using and creating genealogical software was unmatched when compared with anyone else I would have likely found. The user testing was carried out on 7th December 2015.

It should be noted that testing took place on the iPad Simulator running iOS 9 on a 2012 Macbook Pro. This was due to problems with the JSON library I had used in earlier versions of iOS.

4.5.2 Methodology

Significant amounts of preparation went into setting up these sessions over and above the development of the application itself. The project required Level 1 self certification within the Informatics Ethics Regulations and several sheets had to be drawn up for the participants which were testing the application, all of which referenced my ticket number, "[KMRT #810]".

A "Momentree Information Sheet"A.1 was written up to supply information about the application, my motivations and the link to my studies. It also included information regarding the participant's time, rights, benefits, risks, cost and confidentiality. On reading this sheet it was required that the participant sign the "Momentree Consent Form"A.1, agreeing to the conditions. A supplementary sheet was made available to anybody which wasn't comfortable in making their own tree. This suggested they attempt to implement the tree in Figure 4.10 which would exercise every aspect of my software. This however was never used, but acts as a good example as to what people were roughly adding content wise.

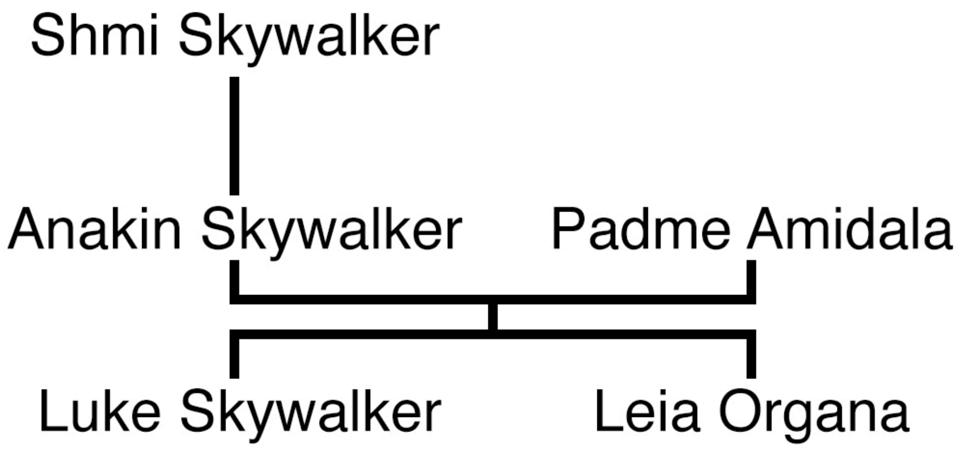


Figure 4.10: Example Tree [9]

Before testing took place, sample Person objects with predefined relations were already implemented within the FamilyView screen. This enabled the PersonList and the associated FamilyTree on the right to be populated. Giving the user a glimpse of the Software's functionality before using it. I feel this gave them a better understanding of what the software was about and allowed them to have a reasonable idea of what their actions would amount to.

Each user testing session was structured as an interview, with myself recording the session to listen back to it later. Initially, I'd suggest to participants what the software currently does and ask them to implement a basic family tree. I requested all subjects implemented their own immediate tree (themselves and parents) using all three screens, FamilyView, FamilyBuilder and PersonEditor. Once the participants had carried out the aforementioned task, I gave them the option of continuing to use the software, adding their grandparents or children. When I was satisfied they had used the software enough, I then asked them a series of questions A.1 alluding to their experience with the software and using their answers, I was able to begin to form the basis of the next iteration requirements.

4.6 Results

A document titled “Momentree Interview Transcripts” A.1 was written up, documenting what was said within the sessions. All the quotes given in the below results are taken from this this document.

4.6.1 General User Interface

Participants knew how to use my app due to its visual similarity other standard iOS applications. I did not have to explain to them how to do, only what I wanted them to do. One of the participants, User 1 remarked “it certainly follows where I would expect things to be as far as other apps I use in iOS”. Another of the participants, User 2 stated “I feel that this would be kind of natively understandable to use on a touchscreen device. I think that flows very nicely”. These remarks were a result of the user interface containing as much standard elements as I could. Giving the participants something they had seen before usability wise in a new context certainly lowered the learning curve.

4.6.2 FamilyView Screen

All participants were not immediately clear however that the application itself was relevant to genealogy. This was something I wanted to ensure going into the session and is why I hardcoded several Person objects with relations into the application. This populated the list on the side, giving the participants people to initially flick through and their family trees to see. One of the participants, User 3, remarked that “it looks very different from other family tree applications” and that "it looks more like brainstorming software". User 2 however stated that "the diagram here with the different people that were presented certainly looked like it was dealing with a family tree sort of diagram". This contrast in opinion was solely a visual interpretation of the FamilyTreeView within the FamilyView screen. The basic tree design was going to be replaced in Iteration 2 regardless, so was not one of my major concerns.

4.6.3 FamilyBuilder Screen

Participants had no problem adding a person simply due to their familiarity with the iOS standards. Pressing the "+" at the top of the PersonList was immediately obvious to everyone. User 1 went as far as stating "well… the big plus button would be my first indication" when asked to add a person. User 3 remarked too that "yeah, once I saw that symbol, I knew exactly what it was for". Once participants had hit the relevant button, they were then faced with the FamilyBuilder screen.

Every single person remarked that the FamilyBuilder screen was initially confusing due to one having to add their parents as a Person before being able to assign them to themselves. This was also evident in my observations in watching them use the software. When adding themselves, they were looking at how to add their parents from the same screen. User 3 remarked that "that's generally how it works, you would add a name, then you add the father, the mother and the children" and that "you have to start at the root. And the root is generally you.". This fact was also echoed by User 1 who stated that doing it this way would "allow you to come out with something a lot faster."

4.6.4 EditPerson Screen

The problem of having to add one's parents before assigning them to themselves caused participants to go back and edit themselves using the PersonEdit screen. This gave all users a chance to actually use functionality and respond to it. On the most part it did respond as expected with User 2 remarking that "I liked that a lot, I thought that was cool" and User 3 stating "that was fairly intuitive. When I saw how it worked, it was no problem. Now I could go ahead and add everybody". User 1 did come across a bug however, stating that "seems like it was unhappy with just having a single parent". This was simply a case of myself not checking every possible scenario in the UI testing.

A participant, User 4, struggled initially in finding the screen suggesting that "I think maybe you should have things that first help people". He alluded to potentially having pointers or a tutorial.

4.6.5 Future Functionalities

When myself and the participants were happy with the amount of time they'd spent with the software, I asked them about how they would imagine generating a photo timeline based on the tree they had built. This was mainly to gauge their reaction to the application's primary -still unimplemented- feature. Almost universally, they expected a button to do this, probably on the right of the screen, near the tree as User 2 stated "if it's closer to the diagram itself, I think it'll be more clear that it's unique to them".

On understanding the photo functionality User 3 remarked "I haven't seen this! The family tree builders are focused on chronologies, records and going back as far as you can go back. They tend not to have a media element to them so this is really useful." This quote was helpful in that it allowed me to assume that people were beginning to fully understand the concept behind the application. What I had had initially was participants assuming this was a 'family tree application' and were getting confused as to why they were unable to add dates and other genealogical information to their added persons.

4.7 Evaluation

From the above user testing, it is possible to look at what challenges the application faces in refining the requirements for the next iterations and beyond.

4.7.1 The ‘Root’ Problem

Without a doubt, the most evident problem as regards to usability would be what I will call the ‘Root Problem’ for the next couple of chapters. This Root Problem appears when a user wishes to add a Person as the root of their tree and assign the parents in one step. What however has to be done in this current iteration is that the parents must be added before their child is created if the user wishes to assign the relation within the FamilyBuilder screen. The alternative however is that the user adds the three Person objects and opens the EditPerson screen on the desired person to assign their parents.

The solution to this problem would be to add the option to add parents as new instances of class Person on both the FamilyBuilder and EditPerson screens. This will form one of the main functional requirements in the next iteration.

4.7.2 Software Design & Implementation Process

While the software performed the main operations well, with only one bug which was detrimental to the user experience, there were several issues here I would like to reflect upon.

The first would be the design. The software was designed to be maintainable and well documented, with the UML class diagrams as I would constantly need to be going back to it due to the iterative design cycle.

The Person class and the FamilyTree class were originally created as one class. Due to the agile development of the project many of the design decisions were taken as the project was in motion. As the functionalities developed, it became apparent that to encapsulate different behaviors better, the class should be split into the two respective classes outlined in the Design section.

The above refactor was made extremely easy in part because of the large behavioral test suite written early on in this iteration. This maintained functional integrity within the code by ensuring that any changes made as result of the refactor did not alter the behavior.

The two controllers for handling the input of Person information, FamilyBuilder-Controller and PersonEditorController were functionally identical with the exception of the addition of a new person. In good software practice, one should have inherited from the other using polymorphism in an effort to minimise repeating code. This however was not done due to my confidence with the new programming language Swift and the short time made available to me through the iteration schedule as other elements such as functionalities took priority.

As I mentioned briefly in the Implementation, the FamilyTreeView was incapable of displaying ancestors and descendants at once and instead resorted to snapping to the largest ancestor in an effort to show the whole tree. This was obviously problematic within the software as a whole, but worked very well for the sake of demonstration in the user testing.

4.7.3 Usability Process

As well as the Root Problem, several other issues came to light throughout the first iteration of user testing. The first and foremost was the way I presented the information to the user.

I was never particularly happy with what became the FamilyTreeView in the FamilyView screen. It lacked the characteristics of what is called a 'Pedigree Chart'[10], the type of diagram generally associated with a Family Tree. This caused confusion with some of the participants in the user testing as they did not initially realise it was a family tree and assumed it was "mind mapping" or that it looked "more like brainstorming software" before they started adding their relatives.

In order for users to figure out how to do something, they need to know what it is they want to do. This is why I believe it is extremely important that in understanding this application as a whole, one must understand the family tree element of it first.

To allow the user to immediately understand conceptually what the application is about, it has to build on characteristics they have previously seen within user interfaces. This is why the standard iOS icons, buttons, pickers and tables were used. What we did have however as regards to the FamilyTreeView was User 3 remarking that "it looks very different from other family tree applications". While it is all well and good to attempt to reimagine the family tree application, that is not what this project aims to do and having something which is unique to the point of it not being instantly recognisable is detrimental to in getting the user to understand the point of this application.

The whole module comprising of the Family Tree Builder currently exists for the users to enter their family details in as quickly as possible such that they can then use the photo functionalities implemented in Iteration 3. Having a new interface which might have a unique, "modern approach" as stated by User 3, adds another layer of figuring out the interface.

This is why I wish to look at this part of the project quite significantly in the next iteration, making it not only more intuitive to use, but instantly recognisable so we can build on the user's assumptions as to what they can do next.

4.7.4 Testing Process

The testing process with NEHGS went well considering the amount of functionalities which had to be cut just to get a usable application in their hands. I perhaps underestimated how difficult it would be for me to get across the concept of being able to generate the photo collection from a family tree, especially with the application in the barebones state it was in.

This resulted in participants prioritising their feedback towards what features my application lacked with respect to a genealogical family tree application (birth, death marriage dates), attributes that were completely irrelevant to this project as a whole. Overall though, I feel my user testing produced good usable data which can carry over onto the next iteration.

4.7.5 Iterative Design Process

Several issues arose with respect to the Iterative Design Process developing this application using.

Factors such as time played a large part in everything looked at previously in this section. From software design quality to functionalities which were cut from this iteration just to have a working application at the testing stage.

One significant problem I had, particularly at the start of the user testing, was that the initial concept of the finished application - using a family tree to generate a timeline of photos - was difficult to get across. I feel this is due to the iterative design model, pushing my application into user testing before all of the main features were implemented. Had the Photo Display Module been added, I feel participants would've understood the application as a whole and given their feedback based on that. That being said however, what the participants did give feedback on was a Family Tree Builder and the pros and cons that were seen within that.

Given that this is the first iteration, it was bound to be the most difficult. This iteration has arguably been done using the Waterfall model of design on a small time duration. Now that the first user testing is carried out however, it is now possible to refine the requirements further and begin to benefit from the positives of the Iterative Design Process, creating what should hopefully be a more functional application with a high degree of usability.

Chapter 5

Iteration 2: Pruning Family Trees

5.1 Introduction

In this chapter I look at the design and implementation challenges I faced in developing my application for the second round of user testing, again with The New England Historic Genealogical Society (NEHGS).

5.2 Requirements

In this section I will look at addressing the problems I faced at the end of the previous chapter as well as adding new functionalities as outlined in Iteration 0.

5.2.1 Functional Requirements

- Extend the family tree builder and person editor to include options of adding new parents as well as selecting existing ones.
- Develop a the family tree diagram to be more in line with existing software.
- Add toggles which allow users to customise the height of the tree diagram in both ancestor and descendant directions.
- Fix bugs arising from Iteration 1 User Testing.

5.2.2 Non-Functional Requirements

- Focus again on usability with respect to efficiency.
- Continue with minimalistic design.

With this new set of requirements, we see a collection of new features and re-iterated features, enough to attempt to do in the timescale, but also enough to hopefully give us a vastly different product with respect to functionalities at the User Testing stage.

5.3 Design

The software design in this iteration is obviously building on what had been done in the previous iteration and I will structure this section similarly, following the Model-View-Controller (MVC) design architecture as to give full rundown of what has been changed and what has been added. In an effort to best convey the alterations, I have changed the order to View, Model, Controller.

5.3.1 View

The view again comprises of the three main screens we are already familiar with, the FamilyView, FamilyBuilder and PersonEditor but also includes a “Moments” screen which will be described in detail below. I will go through each screen referencing any updates and new features with respect to the Requirements. The updated states of the application with respect to screens and transitions can be seen in Figure 5.1.

The design of the FamilyView screen (see Figure 5.2) is the one to have gone through most visual change. Here, I wanted to update the FamilyTreeView to be more inline with a Pedigree Chart as to avoid any confusion as to what the diagram actually was.

The second noticeable change in Figure 5.2 is the box below the list of people with toggles that can control the height of the ancestor and descendant generations. This gives users control over the shape of their tree, allowing them to customise who appears and who does not, paving the way for use of the tree as a search

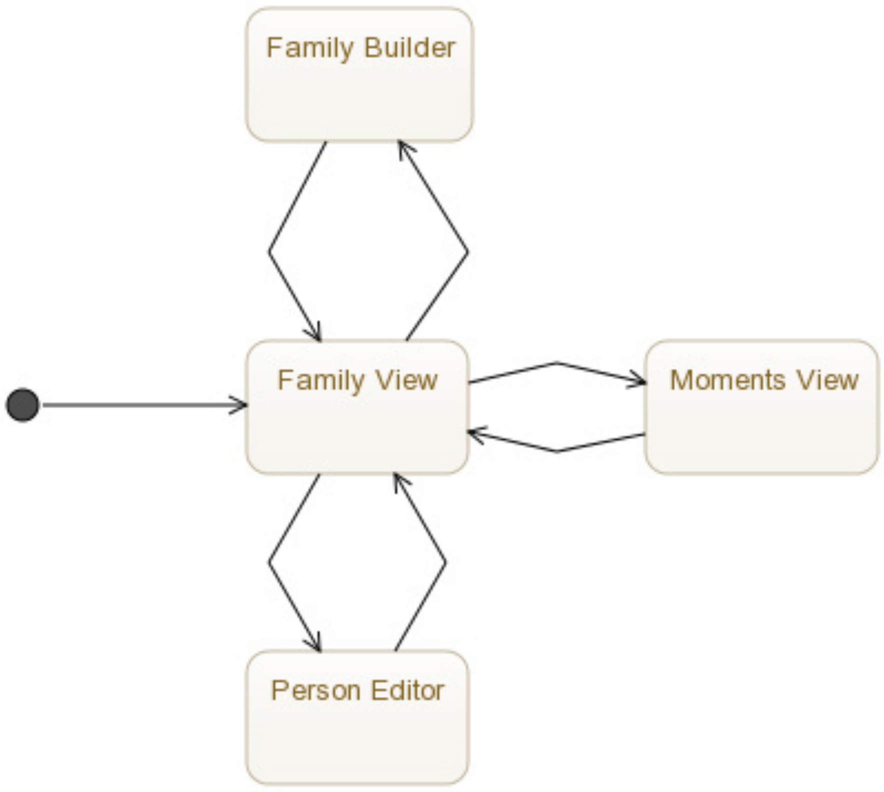


Figure 5.1: Iteration 2: View Transitions

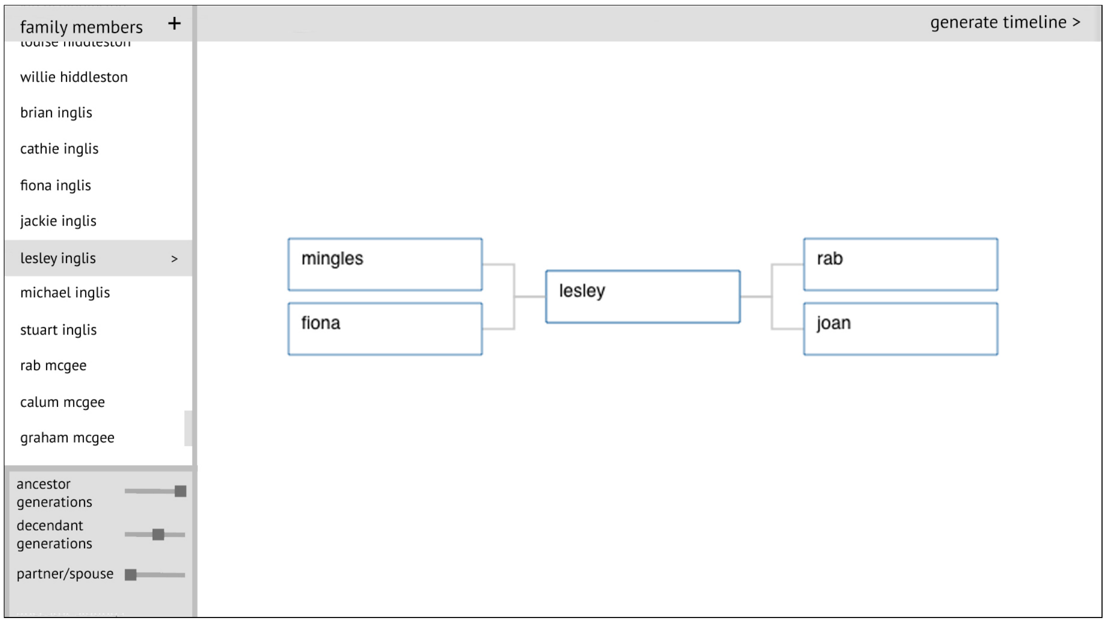


Figure 5.2: Iteration 2 - Concept Render FamilyView

tool for their photos. Aside from the two aforementioned toggles, it also includes a “partner/spouse” toggle for Person at the root of the tree which determines whether or not their partner is included within the tree.

The other views carrying over from the previous iteration, the FamilyBuilder screen and the PersonEditor screen visually remain largely the same so there was not a requirement to create concept renders. Planned alterations to both simply include a toggle which allows them to switch between adding a new person or assigning an existing person as a mother or father.

The “Moments” screen mentioned above was is to serve as a prelude to the Photo Display Module stated in Interation 0. This uses the new functionality outlined in the design of the model below to generate a list of people whose photographic Moments will be drawn. It also was the result of my software design methodology, concurrently researching and designing the new module for the next iteration as I was carrying out this iteration. I will however look at this process in detail within in the next chapter.

5.3.2 Model

The two classes, Person and FamilyTree remain with several alterations. I will begin by looking at what methods have changed to reflect the planned additions within the view. The updated class diagrams for both respective classes can be seen in Figure 5.3.

Starting at the top level, the FamilyTree class, in charge of drawing the chart, takes in several new parameters. It has an ancestorHeight and descendantHeight. These integers determine the amount of generations displayed on the tree for both ancestors and descendants. The newly added “maxAncestorHeight” and “maxDescendant” variables are used by the controller to assign the maximum values of the slider toggles as seen in the FamilyView screen. The values for these are obtained by getDecendantsHeightCount() and getAncestorsHeightCount() within the Person class.

When a tree is being drawn using the “Dict” methods as described in the previous chapter, it however now can use the two “height” variables to determine the size of the drawn tree. I decided to move the getAncestors() and getDecen-

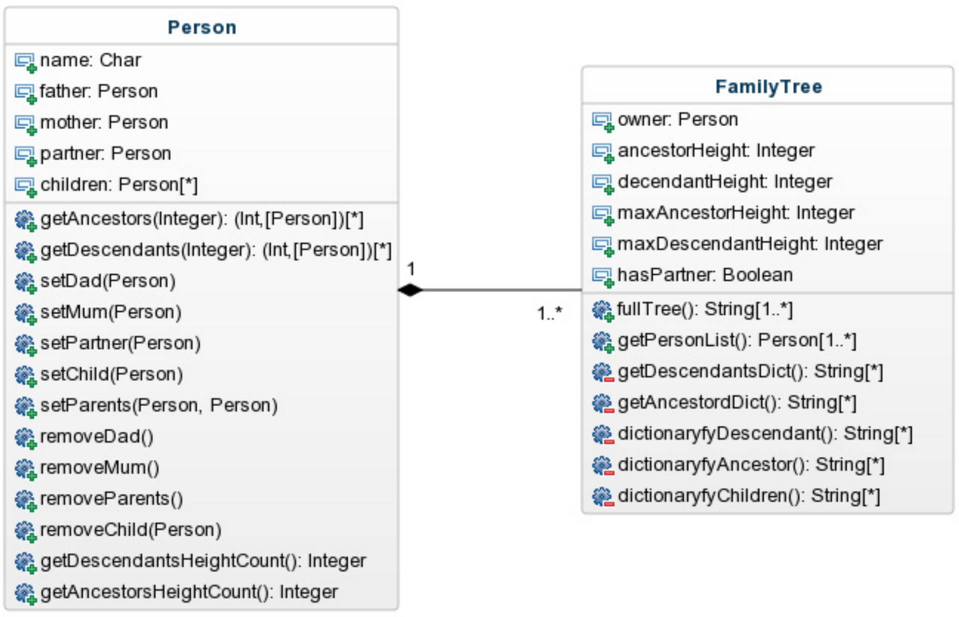

Person
name: Char
father: Person
mother: Person
partner: Person
children: Person[*]
getAncestors(Integer): (Int,[Person])[*]
getDescendants(Integer): (Int,[Person])[*]
setDad(Person)
setMum(Person)
setPartner(Person)
setChild(Person)
setParents(Person, Person)
removeDad()
removeMum()
removeParents()
removeChild(Person)
getDescendantsHeightCount(): Integer
getAncestorsHeightCount(): Integer
1
1..*
FamilyTree
owner: Person
ancestorHeight: Integer
decendantHeight: Integer
maxAncestorHeight: Integer
maxDescendantHeight: Integer
hasPartner: Boolean
fullTree(): String[1..*]
getPersonList(): Person[1..*]
getDescendantsDict(): String[*]
getAncestordDict(): String[*]
dictionaryfyDescendant(): String[*]
dictionaryfyAncestor(): String[*]
dictionaryfyChildren(): String[*]

Figure 5.3: Iteration 2: Person & Family Class Diagrams

dants() methods into the Person class to better encapsulate the functionality. These take in the aforementioned height variables as parameters now and draw the tree based on that. I have included a section of the breadth first search of Person.getDecendants(descendantHeight) in Listing 5.1.

Listing 5.1: Descendant Breadth First Search (BFS)

```
func getDecendants(maxIterations: Int) -> ([(Int,[Person])]) {
    var iteration = 0
    var thisLevel = [Person]()
    var decendantListWithGeneration = [(Int,[Person])]()
    var decendantList = [Person]()

    thisLevel.append(self)

    while !thisLevel.isEmpty {

        var nextLevel = [Person]()
        for n in thisLevel {
            for c in n.children {
                nextLevel.append(c)
            }
        }
        decendantListWithGeneration.append((iteration, thisLevel))
        thisLevel = nextLevel

        if iteration == maxIterations {
            break
        }

        iteration += 1
    }
    return (decendantListWithGeneration)
}
```

What we can see in the code is the BFS terminating when it reaches the value of the height parameter. This gives us only the Person objects within the height of the tree we want.

The getPersonList() method was added in the FamilyTree class for use with the Moments screen. This functionality simply included a list of people that were drawn on the tree and is passed to the controller in change of the Moments screen. This marked the start of the Photo Timeline Module as this method provides the raw input of what is interpreted as “Family Tree” when creating the collection of photos in Iteration 0.

A final addition which should be noted is the hasPartner boolean variable. This simply toggles whether or not the Person’s partner is included in both the fullTree() drawing method and the getPersonList() method.

5.3.3 Controller

The controllers were probably where most of the changes took place. The “Root Problem” was solved by adding new functionalities to the FamilyBuilderController and PersonEditorController. Both switch between two states, a “fatherChangeMode” and “motherChangeMode”, controlled by buttons on the view.

These toggle between the picker view which assigned parents in Iteration 1 and a text box in which a new parent can be added as a new Person. Depending on the state, depends on which input is taken as the edited or added Person's mother or father. The bug which caused a single parent to crash the application was found and the save parent logic redesigned.

The toggles within the FamilyView screen were controlled by the FamilyViewController. This remains functionally similar to what we had before in that it holds and displays the instance of the FamilyTree object. Given that the different toggles were added in the FamilyView screen, we were then able to respond to these within the code. The ancestorHeightSlider and descendantHeightSlider control discrete values between 0 and the FamilyTree object's maxAncestorHeight and maxDescendantHeight. This allows for custom values between the two boundaries which gives different versions of a FamilyTree object to be drawn for the same rooted Person. The hasPartner boolean value like the height parameters also influences the FamilyTree instance output, including the partner as described in the Model section above.

A controller, MomentreeController was written for the Moments screen. This was simply just to display the list of people that the photo timeline will be drawn from as preparation for the third iteration.

5.4 Implementation

The implementation continued off the back of what had been done in the previous iteration.

Given that class functionality was extended and altered, significant amounts of alterations had to be done to the test suite to have it function as required. I had to prioritise development of new code over continuing with a test suite with full coverage due to time constraints.

The main priority in implementing what was outlined in the Design section was the solutions to the "Root Problem". This became problematic as the FamilyBuilderController and PersonEditorController both required it in slightly different formats due the Person being either saved or updated when the screen was closed. In an effort to implement the functionality with minimal refactoring, the same code was written twice in each controller with the required individual customisation. A trade off of good software design in order to have it working before the User Testing.

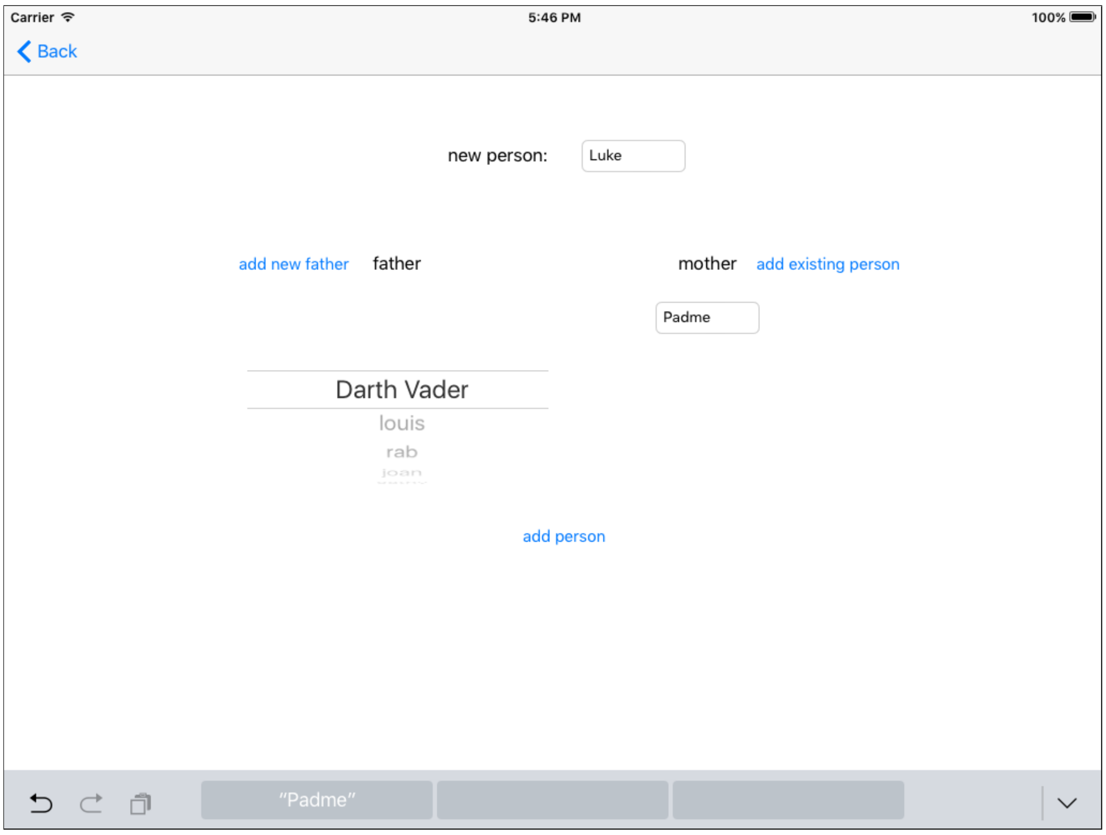


Figure 5.4: Iteration 2: FamilyBuilder Screenshot

In Figure 5.4, it is possible to see the switch in state for how a new mother is added. The button to change the state is titled "add existing" when a new person can be added in a text box and "add new" when the picker is shown to select an existing person. These states can be seamlessly used in any combination and on saving or adding the person, their parents are created as instances of Person if they do not exist and are assigned as partners to each other and as parents to the person in question.

In Figure 5.6 it is again possible to see the Root Problem solution with respect to the Edit screen. The combination of parent addition methods shown is possible due to the bug fix in the adding of single parents.

The FamilyView (see Figure 5.6) arguably underwent the most visual change since the last iteration. What we see is the FamilyTreeView adhering to Pedigree

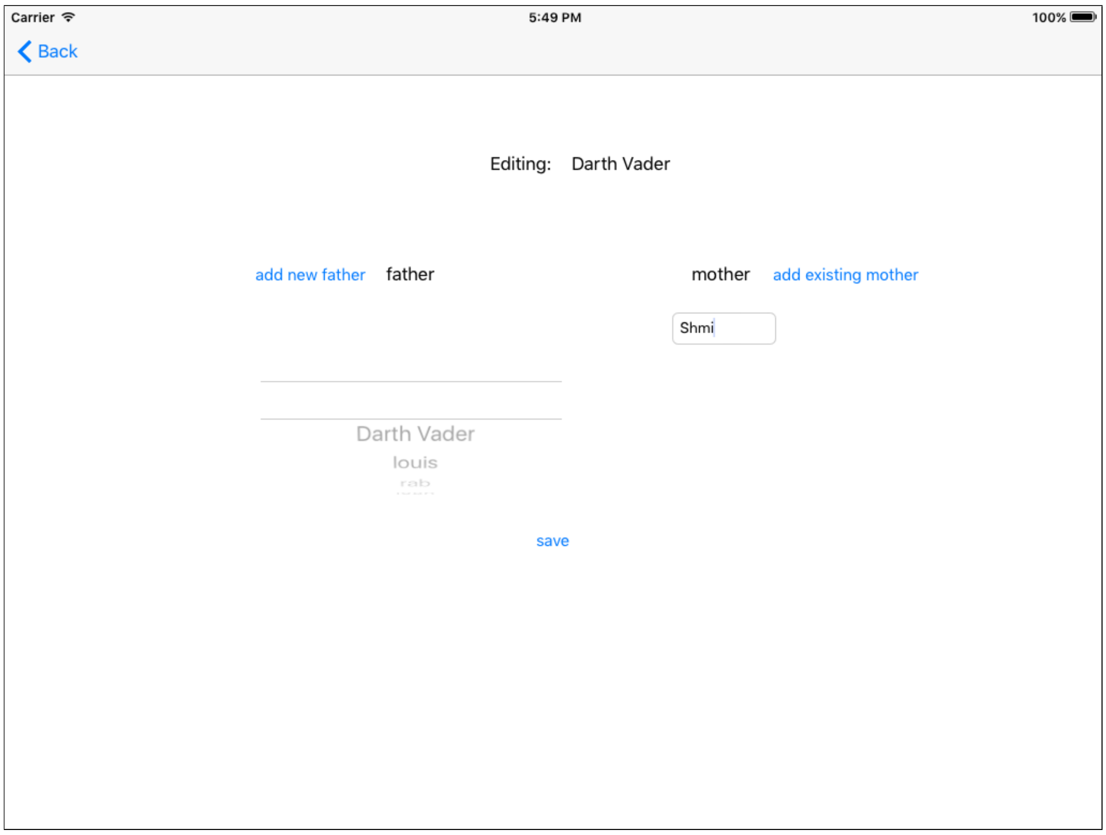


Figure 5.5: Iteration 2: PersonEditor Screenshot

Chart style, featuring names associated with boxes and relations to parents and children clearly defined.

The code behind the First Iteration's FamilyView display was completely rewritten in favour of a new script in the same library d3.js. The JSON information was passed in from the FamilyViewController and loaded the same way but was drawn differently adhering to this new chart script.

While I wrote the code and customised it to suit my inputs and aesthetic output, much of what was implemented from a D3.js tutorial, the source code of which is contained on Github[11]. It should be noted that the problem I had faced of drawing a single tree with ancestors and descendants was solved by drawing two trees for each respective set of relatives, centered around the selected person.

Using this new Family Tree viewer, it became possible to add in elements which increased the dynamics of the view. A time delay which is carried out between updates of the Family Tree allowed an animation to occur as the new tree appeared. On touching the tree, it was possible to move it around and zoom in and

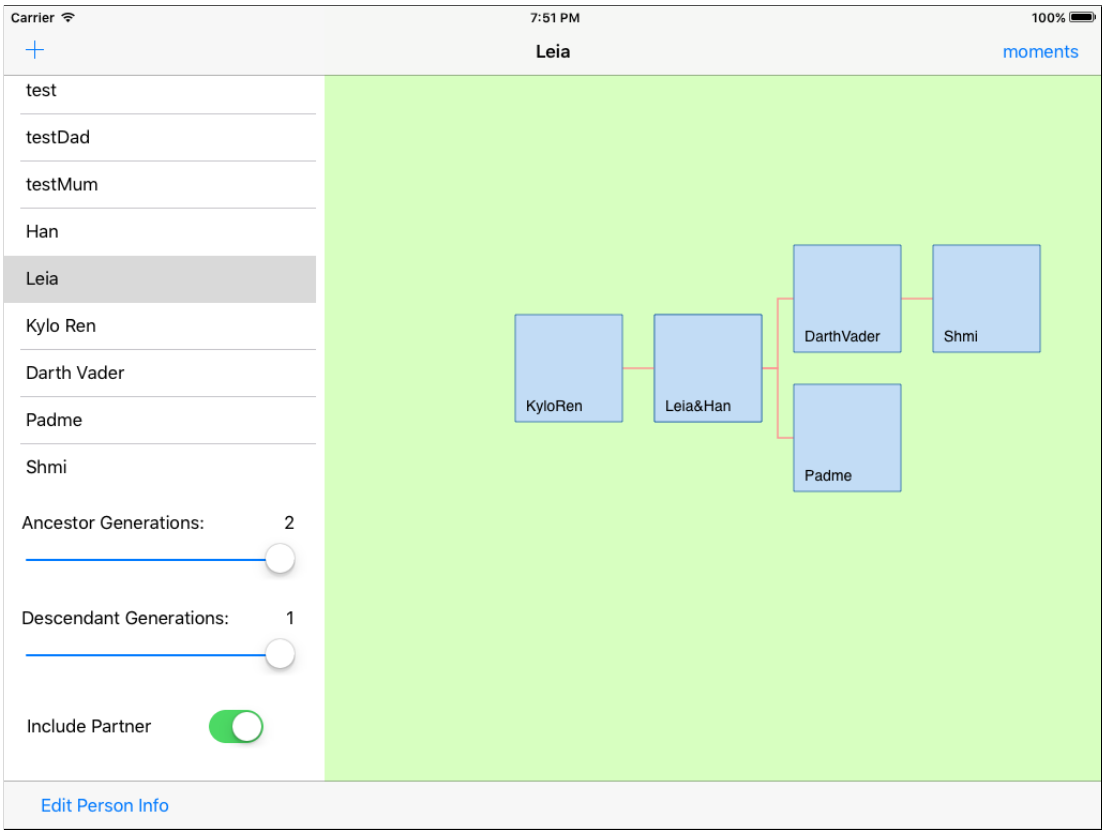


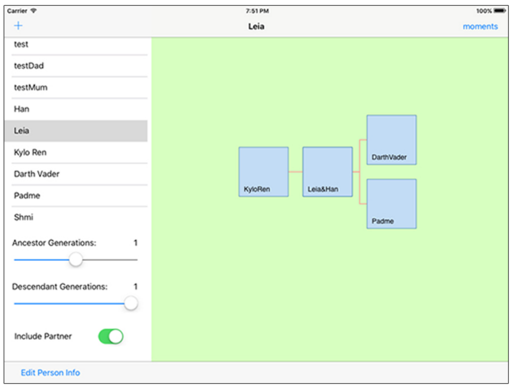
Figure 5.6: Iteration 2: FamilyView Screenshot

out using pinching gestures, familiar with any iOS device user.

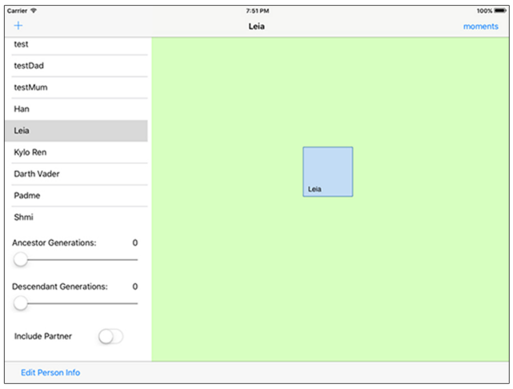
While this and the increased focus on colour and shape contributed very little to functional requirements other than to ensure without a doubt what they were looking at was a family tree, it provided the user with a more tactile experience which was more visually pleasant and enticing.

The effect of the tree customisation toggles can be viewed in Figure 5.7. The text below each of the screenshots represents the state of the FamilyTree object passed to the FamilyTreeView based on the Ancestor, Descendant and partner toggles on the left hand side.

Both the sliders and the switch used to customise the tree again use the iOS standard assets library. On altering these toggles, the FamilyTreeView on the right would instantly update to display the newly drawn FamilyTree instance. In implementing the sliders, I also added a snap to the closest discrete value based on the range of tree heights that were available. This, along with the displayed number above the slider on the right, and the updated FamilyTreeView would



```
familytree(
leia,
ancestorGenerations=1,
descendantGenerations=1
hasParner=true)
```



```
familytree(
leia,
ancestorGenerations=0,
descendantGenerations=0
hasParner=false)
```

Figure 5.7: Iteration 2: FamilyView Customisation Screenshots

give users instant feedback over the size of their tree using a tool within iOS they were already familiar with.

5.5 User Testing

5.5.1 Background

The User Testing for this iteration was again carried out with The New England Historic Genealogical Society (NEHGS) in Boston, Massachusetts on 26th February 2016. It was again tested with five members of their software development team, four of which had used the software in the first iteration of user testing. This was extremely useful in that I was able to reference their exact feedback in the previous user testing and allude to new features which were directly the result of their suggestions.

The testing again took place on the iPad Simulator running iOS 9 on a 2012 Macbook Pro.

5.5.2 Methodology

The same process of preparation went into this session that went into the previous sessions. Another "Momentree Information Sheet" A.1 was drawn up simply reflecting different instructions given the iteration, but ensuring the information regarding the application and the participant remained the same. I again required the participant read the Information Sheet and to sign and date the Participant Consent Form A.1 before starting the session, ensuring everything had reference to my ethics ticket number, "[KMRT #810]".

The application was set up in an identical way as before, featuring family members of my own, with existing relationships to allude to the purpose and functionality of the application. With the permission of each participant I recorded the interview in order to create a transcript from which all the quotes below in the Results section originate from A.1.

The process I asked participants to follow was again similar to the user testing in the previous iteration. I requested they added themselves, their parents and one set of grandparents. This allowed use of all three of the main screens, the FamilyView, the FamilyBuilder and the PersonEditor - even if they added their parents within the FamilyBuilder screen.

Once the relevant relatives were added to the application, I asked the participants to look at the effects of the toggles on both the FamilyTreeView and the names printed to the Moments screen.

After I was satisfied they had interacted enough with the software, I asked them the questions prepared for Iteration 2 A.1. I was then able to use this information to begin to draw conclusions from this iteration to form the requirements in Iteration 3.

5.6 Results

5.6.1 General User Interface

On using the application, I noticed a big improvement in how people were using the features, beyond the fact that they understood what to do having done it with the earlier iteration. User 1 stated “it was a tremendous improvement” while User 3 remarked that “it looks great, the responsiveness is really nice, how snappy it is. It’s neat”. User 2 however said he felt that “the display is very similar to what it has last time”. User 4 interestingly stated “I have an iPad and I find it very frustrating to use. Only because I’m not accustom to the touch screen”. This problem of course hindered User 4’s use of my application and is certainly something to consider in the final Evaluation section in Chapter 8.

5.6.2 FamilyView Screen

The FamilyView screen with the FamilyTreeView was what I got most reaction from. Now that people immediately recognised the FamilyTreeView as a family tree, their knowledge of such systems came out better.

The FamilyTreeView itself garnered a fair amount of response. As regards to how it had changed from the previous iteration, User 1 alluded “it definitely doesn’t have quite the brusk look of a wireframe. It looks a like a little bit more thought has been put into the UI”. User 2 also stated “I liked it!” and User 3, on entering her ancestors said “if I look at Louise now, I can see all her descendants. That’s very cool”. This understanding of the FamilyTree view certainly allowed them to open up on other features, with User 3 going so far as to remark “this is very interesting to me, the ancestor and descendent generations and the way it’s viewed. It’s not something I’ve ever seen before, so it’s novel which I like. It’s intuitive like most genealogical apps or most apps that have some degree of sophistication. You learn it intuitively, it becomes easier once you’ve first started to work with it. That happened in this experience. I like the additions.”

One feature requested by several was for the FamilyTreeView to be interactive as regards to the addition/editing of people and with the list. Several users were attempting to now tap on the FamilyTreeView itself when asked to add data. On this topic, User 4 stated that “I would expect to be able to do things like drag a name into a box or an empty area to create a box and perhaps select two boxes and indicate that I want to connect them as a parent or a child or something like that”. This point was reiterated by User 3 when attempting to edit a person, claiming “that was the once piece where I would want to touch or click on the tree itself. The box containing the person. That’s where I - and you saw I did that. I tried to click on my own name to edit myself". This surprise that the participants had that FamilyTreeView isn't interactive beyond translation and scale is a point which will be explored in significant depth within the Evaluation section.

Another point which came up as regards to the FamilyTreeView which was useful was that traditionally, the descendants were on the left side and ancestors on the right side on a horizontal tree. User 2 stated that "the one thing that struck me as being different is from what I've seen online, they tend to go in the opposite direction" but went onto suggest "that might be what people are used to so they might prefer it. It didn't bother me, it was just different". This was echoed by User 1 stating "I am more used to seeing the root of the horizontal tree being on the left" and User 4 leading on with "I'm custom to seeing it the other way around".

The feedback I received on the customisation of the FamilyTreeView was extremely positive. Particpants appreciated the toggles, User 3 stating "I really like that. I like the graphic element. I like the interface. I like how it moves". User 2 also remarked while using them "I see what that's done. That makes sense", and expanded on his opinion, stating "I thought that was cool. It was a really easy way to navigate the tree, a really cool way of customising it to how you want it to be. I thought that was nice".

User 3 did however suggest the point with respect to the generation sliders that "what people tend to do with any genealogical timeline is go back as far as they can go, so once you get to the point where you have 50 generations, the movement is gonna be a lot more subtle". This however should not be a problem given the application's photographic focus which results in only about 4 or 5 generations worth of photography.

One of the problems I encountered on the FamilyView screen was a failure to update the FamilyTreeView on returning from adding or editing a person. When someone added a parent, the controller didn't update the toggle to include that parent in the tree, so there was initially no feedback as to what changes had taken place. User 1 suggested that "it seemed like that the feedback loop might have been a little bit better in the last round" in response to this problem.

5.6.3 FamilyBuilder Screen

The FamilyBuilder's additions in solving the "Root Problem" were received well. User 1 noted that it was "much more straightforward. Having the ability to make those connections in either directions, that was a lot better". User 3 also commented that "this is much more intuitive. The direction you're going in. You can start from any point and you can add descendants or ancestors". On reflecting on how the application has changed, User 2 remarked "I noticed there was definitely options to add a person as you were assigning relationships, so I think it's something that has become more streamlined." These comments were visualised by me watching them use the software too. Nobody needed prompts or needed an explanation as to how do add their parents this time.

5.6.4 EditPerson Screen

The comments regarding the EditPerson screen itself were stated above with respect to the additions used to counter The Root Problem. What did come up again on discussion of this screen however was that participants struggled initially to find "Edit Person" and their first inclination in how to go about accessing this functionality was from the FamilyTreeView. User 2 struggled in finding it initially and suggested "maybe just make that a little bit more prominent and a little bit more easy to find and I think that would be a more natural way of doing it." User 3 reiterated her comments on the FamilyTreeView, stating again that "I think also it's intuitive to click on the box".

User 1 suggested that "so I guess having something more like autocomplete other than a selector might work" with respect to assigning an existing person. This is an aspect I did consider, but in order to minimise typing on a touchscreen was ultimately decided against.

5.6.5 Future Functionalities

I again gave reference to the photo functionalities would would later be added, explaining it in detail with each participant. Showing them the Moments screen with the list of names led me onto explaining what "Moments" are within iOS and how this app will create a timeline of moments for each of the names seen on the Moments screen. While I was convinced several of the participants struggled to visualise my intentions as the full functionality within the application was incomplete, User 2 did however state that "the idea of working on the moments to draw those pictures in is a really, really, strong aspect of it that sets it apart from similar applications".

5.7 Evaluation

From the final user testing with NEHGS, it is possible to evaluate what still needs to be done to better refine the Family Tree Module to the best it can possibly be as regards to usability in anticipation of the Photo Display functionality added in the third iteration.

5.7.1 The "Leaf" Problem

The major problem I had anticipated in this iteration and in the iteration before was that given the tree takes up two thirds of the screen, it cannot be interacted with in respect to adding and editing people. For the rest of this project, I will name it "The Leaf Problem" because people expect to be presented with this functionality on pressing a leaf of the FamilyTreeView.

This was the reason no FamilyBuilder and PersonEditor screen appeared on the Undergraduate Research Practical (URP)[1]. It was assumed this is how it would be done. On researching however in the first iteration how this might be done, I opted for the webview with an embedded Javascript tree simply to have it finished before the first iteration of testing. While this works well visually in displaying the tree, there is no way to communicate back with the controller once the tree information has been passed in to the webview. This it an issue I will look into more with respect to the iterative design process below.

5.7.2 Software Design & Implementation Process

The quality of software design was less of a priority this iteration simply in order to get the quantity of features and the usability as robust as possible. This resulted in the design problems carrying over from the previous iteration continuing in this one.

While the two classes data is stored in, Person and FamilyTree contained a clean up refactor when their new features were added, the controllers, particularly the FamilyBuilderController and PersonEditorController remained the same, duplicating each others code. This was made worse by the new functionality in solving the root problem, adding more identical code to both.

The completion of the FamilyTreeView however was a large positive step. Having both ancestors and descendants visible at the same time, drawn using the selected person over selecting their highest generation ancestor and drawing all the descendants from there streamlined the software a great deal between the FamilyViewController and the FamilyTreeView.

5.7.3 Usability Process

With the exception of the Leaf Problem, the software generally performed as the user expected. The participants were able to enter in their family trees far quicker this iteration due to the fixes addressing the Root Problem.

The design of the interface here remains as close to what it was in Iteration 1 with the addition of the new features. The buttons and attributes concerning the input of user data remain the same iOS standard toolset, again attempting to build on what the user has seen within other applications.

The design of the FamilyTreeView cleared any ambiguity as to what the application was about. Opting for the Pedigree Chart over what was perceived to be “mind mapping” software however did bring into question more requests for things like relative labels and better defined relations. Given the media aspect of the application, I chose not to add such a feature as doing so would make the FamilyTreeView a “Genogram”[12], a genealogical chart which would require adding further parameters to the FamilyBuilder and PersonEditor screens.

While implementing a genogram would make the tree understandable to users who had no knowledge of the displayed relatives, that is not family, the main demographic of this application. I instead chose to prioritise efficiency over effectiveness as most users will be able to interpret their own relatives in relation to themselves. This point is again reiterated when evaluating the use of the family tree; it is simply an efficient way of input in searching through a photo collection composed of family members. Adding more features in the Family Tree Builder would simply detract from this use case.

Another aspect of the tree that will be looked at resulting from the user testing, is the FamilyTreeView's orientation given that the industry standard is rotated 180 degrees from the current positioning.

A final point which concerned me was the feedback given to the user when making changes to the tree. The ancestor/descendant generation height toggles remained at whatever value they were previously at, so if both were set to 0 and somebody added their father, the father could not be immediately seen on the family tree view. This lack of feedback confused the user and will be looked at when refining the requirements for the next iteration.

5.7.4 Testing Process

The development of the application with respect to the testing process with NEHGS went right down to the wire as before. This resulted in a lack of polish on my part, handing them an application which didn't have features implemented to a finished standard (mainly the feedback problem listed above). That being said however, I was able get far more useful information from the participants this iteration due to the design of the FamilyTreeView making more sense and by explaining the overall concept (including the photos functionality) in high detail. Having done two iterations with NEHGS, I believe anything more with them on this project would be simply be a reiteration of their previous points.

5.7.5 Iterative Design Process

The biggest positive I was able to take from the iterative design process would be the method in which I was able to handle The Root Problem. The contrast of usability between the user testing sessions was extremely evident in both watching the participants use the software and in their answers to my questions about it.

Design elements such as the FamilyTreeView also benefited due to the iterative design process and the feedback in this iteration was again extremely useful, suggesting tweaks, instead of being confused as to what it initially was.

Where this design process however is frustrating would be in the timing; having such short development cycles requires so much to be done to address all the problems in a previous iteration before even looking at new features. Aspects like good software design and test driven development were overlooked in an effort to rush out a usable build again.

Elements such as the FamilyTreeView have also resulted in building on top of what was originally a “quick fix” for the first iteration, resulting in functionalities being potentially limited, a factor I will look at considerably in the final evaluation in Chapter 8.

Chapter 6

Iteration 3: Moments from Trees

6.1 Introduction

In this chapter I look at the design and implementation challenges I faced in further developing my application for the third and final round of user testing, this time with my family and close friends.

6.2 Requirements

In this section, I will look at addressing the problems encountered at the end of the previous chapter as well as adding the Photo Timeline functionalities as outlined in Iteration 0.

6.2.1 Functional Requirements

- Incorporate the photo functionality, providing a timeline of "moments" dependent on the persons appearing in the family tree input.
- Consider concerns raised within "The Leaf Problem" from Iteration 2.
- Address onscreen feedback problems from the previous user testing.

6.2.2 Non-Functional Requirements

- Prioritise only the main functionalities to complete the application use case.
- Increase familiarity through flipping the FamilyTreeView's orientation 180 degrees.
- Make the photos application intuitive like Apple's "Photos".

The new additions regarding the photos and the refined features from the previous iterations give finally what should be a finished application with respect to the use case in Iteration 0. While many features of the Undergraduate Research Practical (URP) have been left out, what will be created in this iteration, serves the core functionality. Any other features are undoubtedly peripheral functionalities, which aim to increase usability, much of which will be looked at and considered in the chapter beyond this one.

6.3 Design

The design of this iteration again builds on the design of previous iterations. Maintaining the Model-View-Controller (MVC) design pattern allowed for considerable maintainability and abstraction of functionalities. This is why I will again divide up this section into View, Model, Controller to outline the added features and alterations that have occurred in this iteration.

6.3.1 View

The screen which holds the photo timeline is known as the Moments screen, adapted from the similarly named screen in the previous iteration. This screen however displays a grid view of photos in chronological order. I created a concept render of the design in Iteration 0 and can be seen in Figure 6.1.

The Moments screen simply displays the photos in a manner we would expect, allowing the user to vertically scroll though the collection and select one based on their preference. On selecting a photo, this brings it up full screen, allowing the user to look at it before returning to the moments screen.

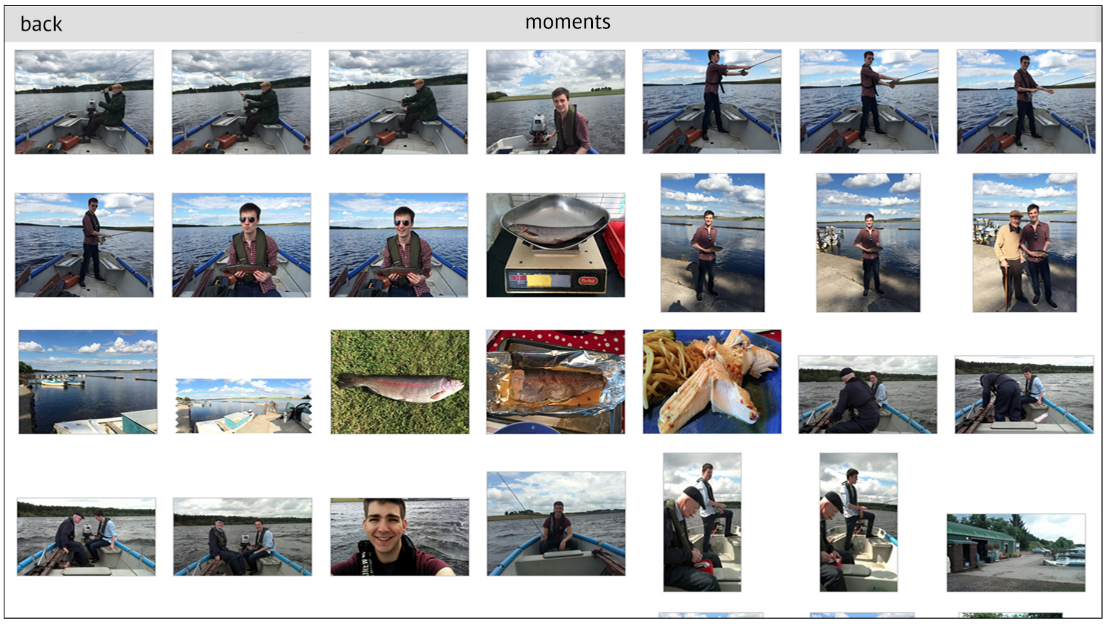


Figure 6.1: Iteration 3 - Concept Render Moments

The FamilyBuilder and PersonEditor screens were expanded to include another picker which selects the album of tagged faces the user wishes to associate with the Person instance in question.

Visually, the FamilyView screen was simply altered to reflect the 180 degree rotation of the FamilyTreeView.

6.3.2 Model

While the FamilyTree class remained the same, the Person object had the attribute “albumTitle” added with the associated getter and setter methods. The final version of the class diagram can be seen in Figure 6.2 and is representative as to how the code currently operates in the finished project.

The album title relates to a unique “localIdentifier” associated with every individual album. The methodology by which this was chosen is outlined in detail within the Implementation section.

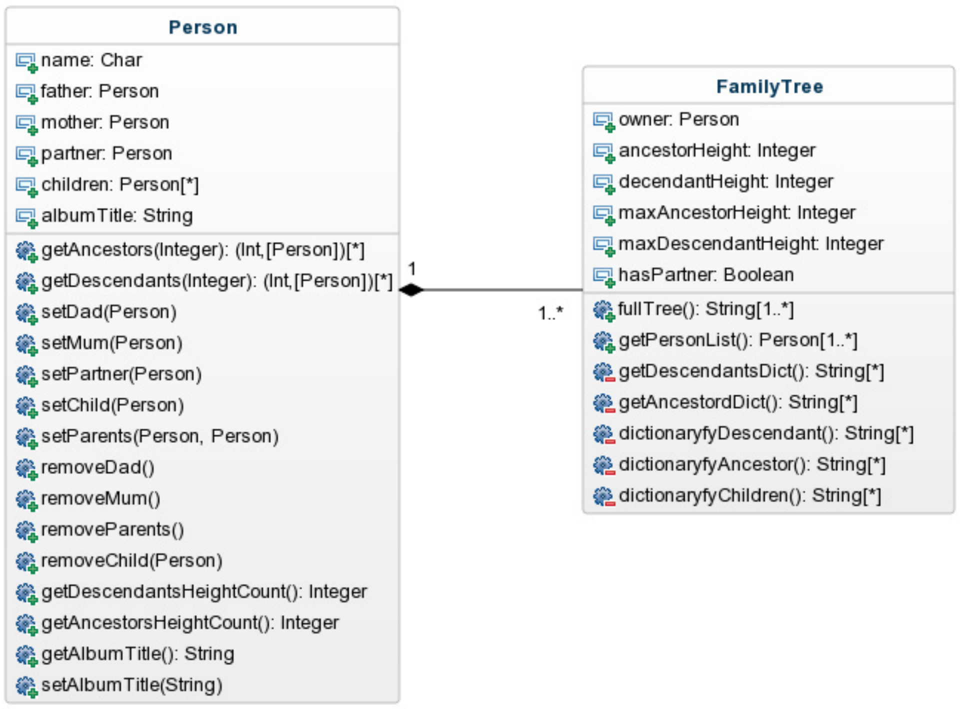

Person
name: Char
father: Person
mother: Person
partner: Person
children: Person[*]
albumTitle: String
getAncestors(Integer): (Int,[Person])[*]
getDescendants(Integer): (Int,[Person])[*]
setDad(Person)
setMum(Person)
setPartner(Person)
setChild(Person)
setParents(Person, Person)
removeDad()
removeMum()
removeParents()
removeChild(Person)
getDescendantsHeightCount(): Integer
getAncestorsHeightCount(): Integer
getAlbumTitle(): String
setAlbumTitle(String)
1
1..*
FamilyTree
owner: Person
ancestorHeight: Integer
decendantHeight: Integer
maxAncestorHeight: Integer
maxDescendantHeight: Integer
hasPartner: Boolean
fullTree(): String[1..*]
getPersonList(): Person[1..*]
getDescendantsDict(): String[*]
getAncestordDict(): String[*]
dictionaryfyDescendant(): String[*]
dictionaryfyAncestor(): String[*]
dictionaryfyChildren(): String[*]

Figure 6.2: Iteration 3: Person & Family Class Diagrams

6.3.3 Controller

The FamilyViewController was updated to reflect the problems experienced with the feedback. Upon returning to the FamilyView screen from editing or adding a person, the generation sliders are maximised to ensure any changes done to the family tree may be seen. In doing this, I also made sure the personList on the left moved up to show the person that was the root of the tree. This allowed the user to see the selected person on the list even if they were initially off screen with respect to their positioning within the list.

As mentioned in the View section above, the FamilyBuilder and PersonEditor screens included the addition of a picker which associated an album of tagged faces with the Person instance. Their respective controllers, FamilyBuilderController and PersonEditorController were used to pull a list of photo albums on the user's iOS device and arrange them alphabetically for easy finding. Upon saving or adding the Person, the chosen album's localIdentifier attribute would be assigned to the relevant person's albumTitle attribute.

The largest addition to this iteration would be the algorithm within the MomentreeController, controlling the Moments screen. In order to form a bridge between the family tree input and what images were displayed on the output, I had to come up with a way of displaying the relevant moments. Due to the length of the actual implementation using the Photos Framework convoluted efficiency tradeoffs, I will simply reference the steps of the algorithm in Swift in Listing 6.1, adapting the objects and methods for readability.

Listing 6.1: Momentree Algorithm in Swift

```
var moments = [Moments]()
for person in familyTree.getPersonList() {
  for photo in photoCollection.getAlbumWithID(person.getAlbum()) {
    let moment =  photo.getMoment()
    if moment not in moments {
      moments.append(moment)
    }
  }
}
timeline.draw(moments)
```

In a further attempt to visualise this information given its importance to the functionality of the application, I have drawn a UML Sequence Diagram to explain the relations between the objects. This is seen in Figure 6.3.

The algorithm I developed stands out as unique as it is simply not displaying a collection of photos with the tagged faces of the given Person instances. It retrieves photos taken in similar places at similar times to the photo with within the given person's "Faces" album.

6.4 Implementation

The changes and additions were again developed in Swift on top of the current codebase. With this Iteration, the Test Driven Development that had occurred before went mainly neglected simply due to the fact that the application was in what developers call "Crunch" time. Most changes were tested visually and the

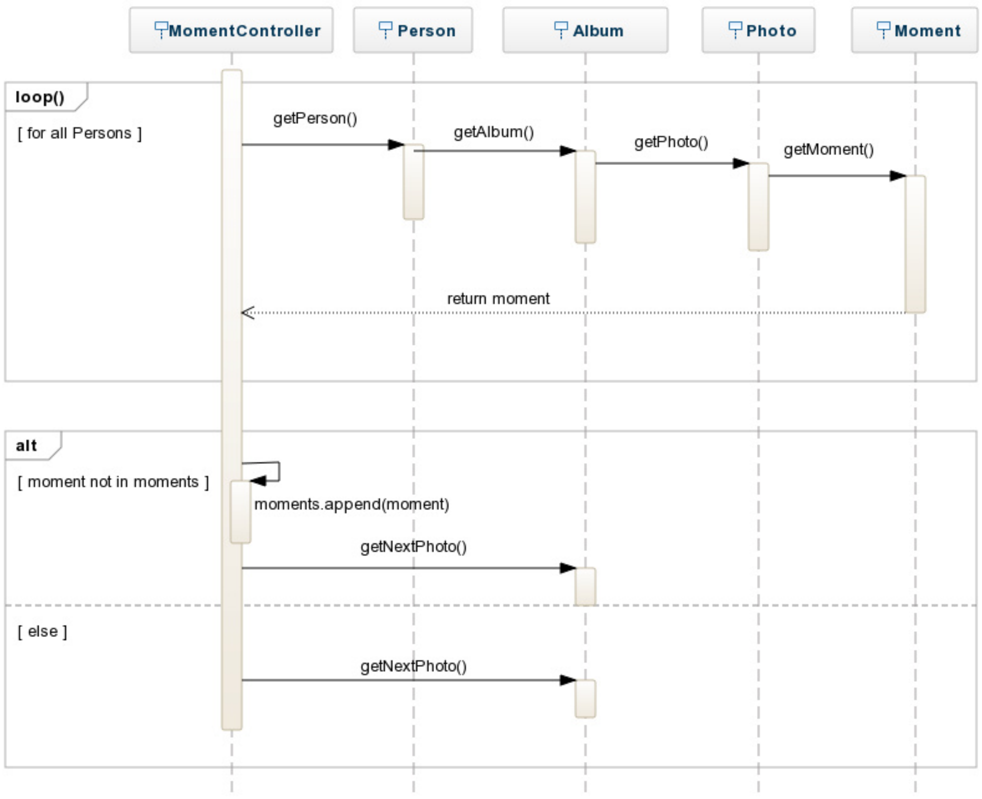


Figure 6.3: Momentree Algorithm Sequence Diagram

code base was stable enough from previous iterations for very few problems to occur.

In an effort to get onto the development of the Photo Functionality, I will quickly outline what other changes took place on the application itself.

As mentioned in the design section, a further picker was added to the Family-Builder and PersonEditor screens. This is available to see in Figure 6.4 and Figure 6.5. While I had originally opted for a table, similar to how the personList, seen on the left of the FamilyView screen, the picker was ultimately selected simply for efficient implementation and familiarity within the application.

The FamilyView screen's visual additions simply include the inversion of the FamilyTreeView and the snapping of the descendant & ancestor generation sliders to their maximum value when returning from the FamilyBuilder or PersonEdit screens.

The extent of the development in this iteration took place using what is called

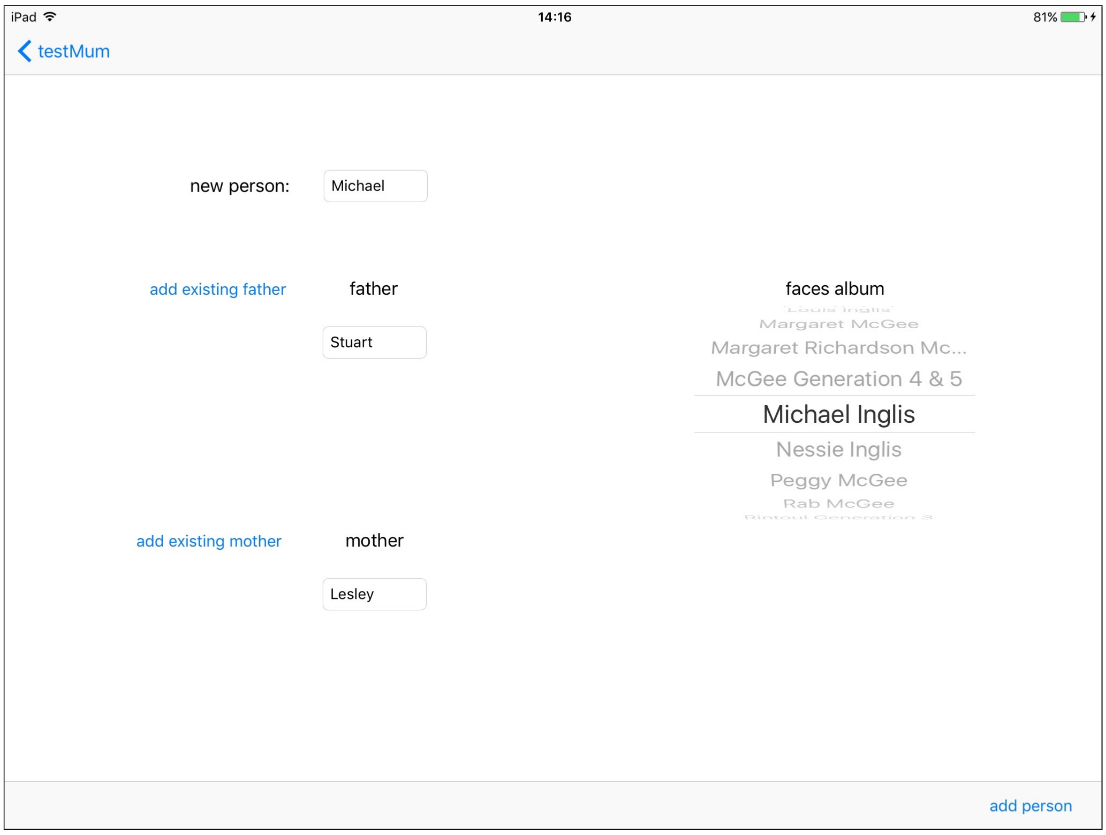


Figure 6.4: Iteration 3: FamilyBuilder Screenshot

the “Photos Framework”[3]. Using the Photos Framework, I was able to create a screen similar to that seen within the native “Photos” application on iOS.

Research of this part of the application started during Iteration 1 due to the extensive framework that needed to be learned to carry out the “Momentree Algorithm” outlined in the design section. Given that the framework only released in 2014 and has only a niche point of interest, material and examples using the framework was extremely sparse. What Apple did provide was the source code for “Example app using Photos framework”[13]. This however was in Objective-C, Apple’s iOS language before Swift, something I had no experience with. What I did find though was a repository of a user, “ooper-shlab” who had translated it to Swift[14].

This example application was crucial in me learning the framework in time to implement my own functionalities. I was able to use and adapt to my needs a couple of the controllers within the Swift Translation. This saved me a lot of time in learning how to populate iOS tables with images and allowed me to concentrate

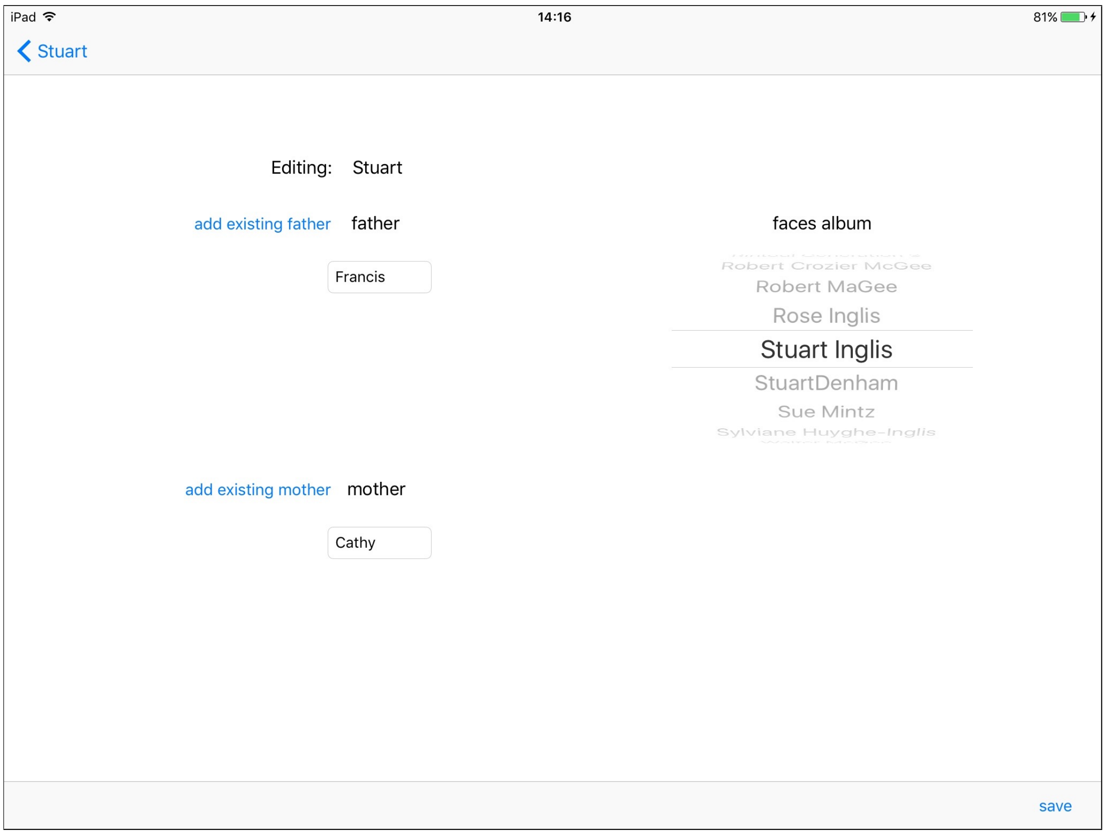
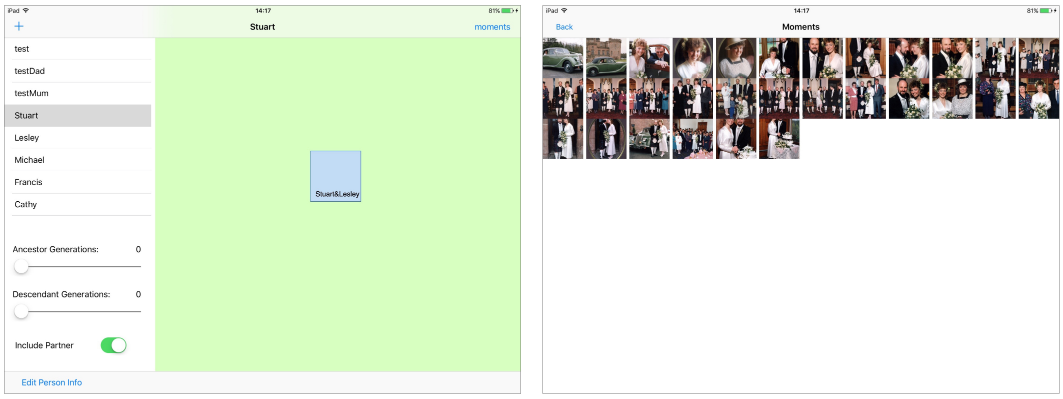


Figure 6.5: Iteration 3: PersonEditor Screenshot

on the main "Momentree" functionality of the application. All controllers taken from this sample application have reference to the copyright owner at the top of the file.

I was able to implement my MomentreeController with both the Asset Grid View (the cells drawn on the Moments screen) and the Asset View (the enlarged photo shown by tapping a cell on the Moment screen) within the Swift Translation to create an application which fulfilled the use case listed in Iteration 0. I have included screenshots of the Moments screen and their corresponding FamilyView screen with altered toggles in Figures 6.6, 6.7 and 6.8. Individual photos can of course be enlarged by tapping them.

We can see the photo timeline changing based on the included people. In Figure 6.6, we see a wedding of Lesley and Stuart. Note that all the photos of the wedding appear, even if Lesley and Stuart are not in the photo. Figure 6.7 exands Lesley and Stuart to include their descendants. This adds new photos of their descendants after their wedding. In Figure 6.8, we see that Stuart's

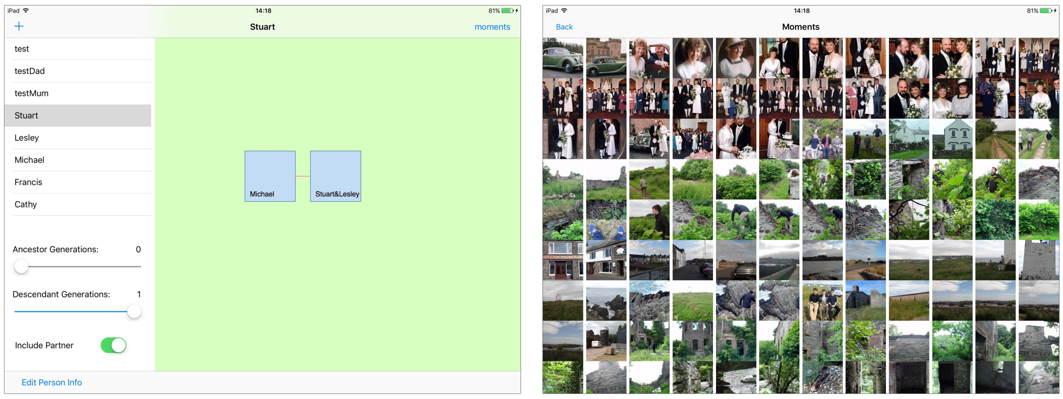


Ancestor Generations: 0

Descendant Generations: 0

Include Partner

Figure 6.6: Iteration 3: Moments Screenshot

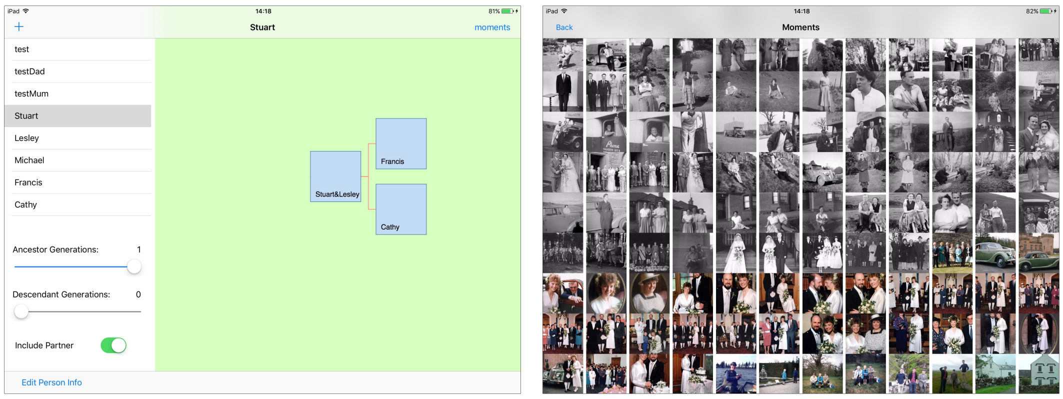


Ancestor Generations: 0

Descendant Generations: 1

Include Partner

Figure 6.7: Iteration 3: Moments Screenshot with descendantHeight=1



Ancestor Generations: 1

Descendant Generations: 0

Include Partner

Figure 6.8: Iteration 3: Moments Screenshot with ancestorHeight=1

ancestors have been added, adding the large quantity of black and white photos of his parents before his wedding.

Dynamic timelines of Moments can be drawn based on the persons that feature within the customisable FamilyTreeView. This was the functionality I set out to do, and this is finally what I did.

6.5 User Testing

6.5.1 Background

The user testing was carried out with friends and close family. The reasoning behind this was the relevance of the dataset I had to test it with. In order for the "Moments" functionality to be properly shown, users had to create and assign photo albums to people they knew.

Given the fact the software takes into account photos within the same Moment as ones in which people are tagged, this increases significantly the work which must be done in sourcing photos to create a valid dataset. This is why I chose to use my own photo collection as the data source. The fact I have photos of relatives stemming from the 1800s, sorted using both date and location - in essence a manually created Moment - was extremely useful.

Using this dataset, I was then able to use one family members and two close friends to user test this third and final iteration on 21st April 2016. The testing was finally carried out with an actual iPad Air running iOS 9.3 synced with relevant data from my photo library.

6.5.2 Methodology

The user testing sessions were conducted in a similar manner to previous iterations. I ensured the participant had read the Momentree Information Sheet for the third iteration and signed the Participant Consent Form A.1 before carrying out the recorded user testing and subsequent interview using the questions written for Iteration 3 A.1.

A single, three member "test family" was set up before user testing as was in previous iterations. To avoid confusing participants however, this time the names were irrelevant to my family as to ensure there were no duplicate names.

During the user testing I suggested names of people to be entered and their album to be assigned (usually evident based on the family member's name). The tree I asked them to create included myself, my parents and my paternal grandparents. This set of Persons and their respective Moments are available to see in Figures 6.6 through 6.8.

I also asked each of the participants to complete to do a System Usability Scale (SUS) questionnaire [15] on completion of the session. I will look at the results below.

6.6 Results

6.6.1 General User Interface

The participants generally liked the simple user interface. User 1 stated that "it's good that all the buttons that are clickable have a similar design, so they're all in blue. And the buttons that don't have a descriptive enough symbol, you've just left as text, which is good" and User 2 added that "it is very clean. Everything is very intuitive".

User 1 also went onto explain that "I like the minimalist design overall. It's very clean compared to other programs I have used which are very dense. It's very easy for a new person to come into this app and know how to use it compared to the other apps that have so much clutter, where there's a bigger learning curve". This point of a learning curve was also reiterated by User 2, remarking that "everything's laid out nicely. I wouldn't complain about that I know what it does. And I think if you would have just given it to me, I would have been able to work out, given a brief description, everything I needed to do to create it. It's pretty simple." This was stated after User 2 remarked "I keep getting confused" as he was using the software. User 3 also remarked about the learning curve, saying "I think it's just getting used to an app like you would have to get used to any new app".

On the subject of functionalities, User 2 said "I think it's just in early stages, I think it's lacking a little bit" before explaining that "I just feel like there's just not as much as you'd expect to be in it. I suppose all you need to do is add people and add photos sort of thing".

Regarding the overall concept, User 3 remarked "well I like the fact that it's really different. And the fact that you can add everybody that's linked to you. It's just nice to be able to tie everyone's photos together just by simply adding their profile to it." User 2 also commented when understanding the overall functionality, stating "right so you make this, then you give it this and then it generates all the photos in the order given the people. Oh okay cool."

6.6.2 FamilyBuilder Screen

Participants again had no trouble adding people to their trees. User 1 stated "well it's clearly marked with a plus sign which is usually a metaphor for add. So I'm guessing I just press that" when adding relatives. User 3 added that "it's very easy. It's very simple, very straight forward".

On the topic of assigning faces to a person, User 1 said "yeah it's pretty obvious. You're connecting up a person stored on the app with a photo album. You've got labels as well, so it's really easy to know what's going on here." User 3 also had a similar experience going onto say "yes because it's saying "faces album", it's identifying that person's face with their album. And I see there's no duplication of names, and the names are all there if you obviously want to identify them."

6.6.3 EditPerson Screen

User 2 had a similar problem to what NEHGS had experienced in previous iterations regarding the "Edit Person" functionality, remarking that the button "was a bit hidden."

On asking participants about this functionality, many of them stated features of The Leaf Problem with respect to the FamilyTreeView on the FamilyView screen. User 3 remarked "it would probably be easier if you touched on the person, and that was identifying them for editing". User 1 reiterated with "the JavaScript on the right should be clickable. So you could click on these people because it's intuitive for someone to want to press on that on an iPad". His comment was expanded to also involve the addition of a person: "the only additions I would have would be a drag and drop. So you could select someone on the left, drag them and it would automatically add them. Cause that reduces what the user needs to do to complete that action." User 2 also stated that "I think it's just cause it's like, so big, you kind of think you can interact with it by touching the actual thing."

6.6.4 FamilyView Screen

Comments beyond those relating to The Leaf Problem as stated above, surrounded mainly the ancestor/descendant generation toggles.

User 3 liked that "it switches them on and it switches them off. Pretty straight forward", going onto say that "I like the fact that you can include the partner, in or out". User 1 also stated that "the sliders are nice so people don't have to manually enter the number, and obviously it will only give you what you can only slide it to, which is also nice."

As regards to the newly implemented feedback updates on returning to the FamilyView screen, User 1 stated "because of the animation you can see what's changed, and what's remained the same." The numerical feedback beside the toggles caused User 3 to comment "the fact that it's immediately telling you there's all those generations there. You've just got to sit and read it and look at it."

6.6.5 Moments Screen

The Momentree search functionality was what I wanted to get most feedback on in this iteration.

On finally generating a list of Moments based on her selected family tree, User 3 remarked "it's great and the fact that you've got from wee boy to very old man. It's got everything in here." User 1 also stated at this point that "yeah it's pretty nice. There's no other way I can think of organising it." User 2's remarks on doing this were "alright, that's pretty cool. Yeah this is quite different, but it's nice to have it, you can follow a family through", adding "it was nice to have it in chronological order".

A comment shared by participants was that they thought this photo timeline could be made better showing some form of separation between the moments associated with different moments. User 2 started with “maybe if there was some clear indication that this was you changing from generation” before going onto say “maybe some sort of visual sign, like a line saying there’s a new generation now, and a new generation now. That would have been quite cool. Or some sort of background color the person can use to kind of indicate what family members are where”. User 1 also suggested similar, stating “uou could maybe have a way of labeling which person is related with the photo. I don’t know if that might be useful because looking through this I might want to know whose photo it is”.

6.6.6 SUS Score

The result for each of the SUS scores as follows: 83, 92, 85, giving an average of 86.7.

6.7 Evaluation

6.7.1 Software Design & Implementation Process

The design and implementation of this iteration underwent the least test driven development thus far. A problem of the Moments functionalities required that real data be used to test the application, something which would be impossible to replicate within behavioral driven test driven development.

Features worked as expected, with no bugs appearing this iteration. None of the software design problems which required refactoring were considered as to keep the code base stable in anticipation for adding the new functionalities within the timeframe that was available.

The fact the Momentree functionality was able to be implemented in time was a monumental achievement within the software development, completing the main use case for the application, outlined in iteration 0. This now allowed for any new suggestions within the user feedback to be regarding increased usability or peripheral functionalities for this application as a whole, something I struggled with in previous iterations.

6.7.2 Usability Process

A unique aspect of the testing process in this iteration was that the software performed exactly as expected. There was no bugs and problems with respect to feedback which hampered the user experience overall. This maintained focus on the functionalities of the application, allowing participants to think about how it could be made better rather than whether or not it was functioning as it should.

An aspect of the user interface which was mentioned several times in the User Testing section was the learning curve. The learning curve I feel is pivotal to not only how the user uses the software but whether the user uses the software. This is why I put a lot of my energy into an extremely basic user interface design, constantly using repeated elements for previously seen functionalities such as transitioning between screens or data selection. Elements of the platform also played a large part in this, selecting elements which favored tapping and scrolling over typing on the touchscreen. Attempting to lower the learning curve was crucial in getting over the initial hurdle of the user entering in their family tree before being able to use the application's main functionality.

The Leaf Problem again surfaced in this iteration. I will however explain my evaluation of this in depth as regards to its continued persistence in this iteration within Iterative Design Evaluation below. Users with no genealogical software experience had no trouble in understanding what to do with respect to the Family Tree builder, with no ambiguity around the FamilyTreeView.

There were several issues with usability as regards to the Moments screen itself. The one I predicted such as not being able to swipe through photos upon expanding one to full screen came up. This is an intuitive functionality synonymous with photos on a touchscreen device and while it was no detriment to the application's main functionality, it stifles the user's efficiency by not being included.

The most interesting thing to be brought up about the Moments screen would be the suggested segregation of generations. This would add a large amount of visual feedback with respect to the photo collection. Allowing the timeline to be better interpreted adds to a couple of the core aspects of this application; organisation and display.

A problem arising from this however would be how to handle such a thing. Suggestions included using colours as backgrounds on different generations. This would also require labels outlining the significance of each colour. We would additionally be tasked in dealing with the problem of relatives of different generations overlapping within moments. This however could likely be solved by declaring the cut off for each partition to be the first moment in which the new generation appears.

6.7.3 Testing Process

The feedback from the user testing was extremely useful. Given that the application was finished, people were able to consider what features they would change with regard to the full Functionality. This gave rise to suggestions such as the generation partition on the Moments screen.

Having the participants knowing the family members they were adding was also crucial in them understanding the photos which appeared on the moments screen. This could only have been carried out with close friends or family as other participants may not have been able to determine who was appearing within Moments.

In putting together the dataset of photos, I also took into consideration the photos that were included for each of the generations (myself, my parents and my grandparents). I ensured they contained distinguishable events such as weddings or birthdays. This was helped by the fact also that for the grandparent generation, much of the photos were black and white, easily distinguishing them as older, allowing the timeline to be interpreted as an ascending chronology.

An aspect that arose with regard to the testing device - the first time the iPad was used over the simulator - did however bring up an issue within the UI that don't occur previously - the keyboard taking up half of the screen. All participants were however familiar with the iPad, so dismissing the keyboard to move onto the next text entry wasn't an issue.

Getting the SUS score allowed me to benchmark the usability of this piece of software. I look into the implications of this score within Chapter 8.

6.7.4 Iterative Design Process

For this iteration, the iterative approach to development, forced the application in the direction of prioritising core functionalities over peripheral functionalities. This was problematic to the overall usability, but ensured that the application actually was finished within the time available, allowing one user testing session with the completed functionality. I would say that I successfully carried out the functional requirements in this iteration, as the Leaf Problem remains.

The Leaf Problem was not addressed in this iteration due to the amount of work that would be required to bring about the changes suggested by users. The design of the webview system was implemented in Iteration 1 and was changed only visually in iteration 2 due to time constraints within the development cycle. To implement a responsive family tree builder would have required the system to be built from scratch in the first iteration, not something that would be possible to do in the third. This issue will form one of the main arguments against this development methodology in Chapter 8.

Unlike Iteration 2, where the bulk of the iteration's development was building on what was refined from iteration 1, this iteration dealt largely with new features, albeit with minor usability issues remaining from Iteration 2. For the application to feel completely finished in the hands of a user, one more iteration would take place, similar to how the second iteration handled the first, refining the usability within a "raw", new feature.

Given the nature of this project with respect to the iterative development, the next chapter will outline what requirements would carry over from this chapter had there been another iteration within the time given. This will outline conclusions from this chapter with respect to the development cycle, providing more substance for an overall evaluation of the iterative development cycle's impact on this project in Chapter 8.

Chapter 7

Iteration 4: Maintaining A Forest

7.1 Introduction

In this chapter I look at how the feedback and evaluation from the user testing session in Iteration 3 would result in a fourth iteration's requirements.

I will also explore what further features would be considered beyond a fourth iteration, much of which was thought of in the early weeks of the application development.

7.2 Future Requirements

In this section I will look at addressing the problems I faced at the end of the previous chapters and how, developing them in a fourth iteration would increase the design and usability of the software by building on the core functionality as it is at the end of iteration 3.

7.2.1 Functional Requirements

- Refactor FamilyViewController and PersonEditorController, using polymorphism to remove repeated code.
- Clean up test suite to reflect changes in code structure and to maintain code integrity.

- Add an "Onboarding Experience".
- Moments Generation Partitions.
- Address usability issues regarding the expanded photo.

7.2.2 Non-Functional Requirements

- Prioritise usability and code maintainability over new functionality.
- Increase familiarity by involving intuitive touch screen gestures.
- Continue to liken the photos timeline to Apple's "Photos".

7.3 Future Design & Implementation

Given that the above features are not - and will not - be implemented, the amount of detail regarding their design will be minimal. This section is simply to explore how each of the respective new requirements, derived from previous iterations use the iterative development cycle to increase the quality of the software's design and usability.

7.3.1 Codebase Refactor

A refactor of both the FamilyViewController and PersonEditorController, encapsulating both of their shared functionalities in an abstract class would go a long way in making the codebase more maintainable for future additions. It would drastically cut down the amount of lines of code within the project and ensure shared functionalities in both always match each other.

Additional clean ups within the project which are required include the removal of unused methods within the classes and the streamlining of some processes which were written as the framework they use was being learned.

7.3.2 Updated Test suite

Before any of the changes outlined in the refactor above took place, a significant amount of work would be required to update and maintain the test suite as to maintain functional integrity across the software. Additions to this stopped during Iteration 2, so much of the changes involving the Person and FamilyTree classes would have to be taken into account and updated.

Behavioral Tests may now be written for the whole sequence of functionality the application now has, given that the photo timeline functionality has been added to the family tree module.

Implementing both the tests and above refactor would allow for the design of the software to be far more maintainable and would reduce the time taken to implement any new functionalities. The time taken to implement these and the length of the iteration would have to be taken into account however. This factor was what previously stopped such requirements being carried out in previous iterations.

7.3.3 An Onboarding Experience

An onboarding experience was suggested by several at The New England Historic Genealogical Society (NEHGS) during the first and second iterations in the interview transcripts A.1. User 2 (iterations 1 & 2) suggested "I know having a first launch onboarding experience, some people think of that as UI cheating, but having something like that would be very useful".

This is certainly a process which would benefit the overall usability of the application. Having such a tutorial would speed up significantly the learning curve explored in the previous chapter. Allowing popups which hand hold the user through loading in their first few relatives before displaying the moments would show them all of the features of the application in a very short period of time. Beyond this, users can then input data using their own method. This would also help in showing the motivation behind the application, something all testing participants required.

Many libraries for such a thing exist in Swift, so developing such a feature would be straightforward.

7.3.4 Moments Generation Partitions

Adding partitions to the Moments timeline, splitting each generation would require a reimplementation of the "Momentree Algorithm".

In Apple's definition of the Moment seen in Iteration 0, it states that "a moment year groups all moments in a calendar year"[3]. What is described as a "Moment Year" could be used to contain each generation of moments. Having the all the photos contained in this file structure would then require the asset grid view controller to be completely rewritten to compensate for that.

Having this functionality would in essence make this application a clone of the native iOS Photos application, whereby "moment years" (the most zoomed out layer of the photo collection) is replaced with "generation number". Usability would be increased dramatically as it would allow the user after generating their photo timeline from their family tree to then navigate it by generation. This would result in an impossible amount of work for one person however, and one would likely have to resort to instead just changing the colour of the background on each respective generation if this was to be implemented in a timescale similar to what has been seen before in previous iterations.

7.3.5 Swiping Photos

The addition of a swipe to navigate through expanded photos on the photo timeline, would allow for usability to be significantly increased. Not only does it take the current interaction count of two to change the photo (go back and select new photo) to one (swipe left or right), but such a swiping interaction is synonymous with navigating through photos on a touch screen device, allowing the user to repeat actions they have previously learned on other applications. This likely could be implemented with minimal alterations to the code.

7.4 Further Implementations

In this section, I want to look at features that were cut in the development of this particular application and in the deriving of the requirements from the Undergraduate Research Practical (URP).

7.4.1 A Solution to The Leaf Problem

This is arguably the most frustrating element of this project. The current functionality would have to be completely replaced to solve this problem.

The webview implementation of this not only prevented the user adding and editing relatives using the FamilyTreeView, but prevented images of the relatives being used on the tree itself as seen in the concept FamilyView screen within the Undergraduate Research Practical (URP)[1] (see Figure 3.1).

Developing this system from scratch within Swift would allow the aforementioned functionality to be implemented utilising the FamilyBuilderController, PersonEditorController and Photos Framework to allow the user to assign an avatar to each individual relation on the tree.

7.4.2 Cloud Functionality

This feature was considered as a "Third Module", on top of the family tree and photo timeline modules derived in Iteration 0. This was cut in refining the requirements for iteration 0. In the URP, I consider iCloud, Apple's online cloud service when recommending a platform:

"It also allows for different users and family members to be easily connected, adding the ability to share and dynamically manage different Ëvents.̈"

Having the user able to share their timelines and trees with other family members would have gone a long way in fulfilling the title of the URP, "Managing and Preserving Personal Photo Collections for Future Generations". What has been created however is an application that "Manages Personal Photo Collections for Future Generations" by creating a way by which collated photo collections of the users ancestors can be displayed. Having photos hosted in the cloud, spread across different devices would however have fulfilled the “preservation” of these photos.

Chapter 8

Iterating Over Each Branch

8.1 Introduction

In this chapter, I will look at how the project as a whole has responded to the iterative development cycle. I will evaluate how the reasoning in Chapter 2 in picking the iterative design methodology has impacted the overall development, specifically the software design, usability and requirements.

8.2 Software Requirements

8.2.1 Positives

The Iterative methodology benefited the formation of the requirements hugely. Not only did the results of testing at each iteration have impact in refining the requirements to suit the user and increase usability, but it forced the development at each stage to prioritise core functionality over peripheral functionalities.

8.2.2 Negatives

As mentioned above, many of the peripheral functionalities had to be set aside in order to have main functionalities ready by each iteration's user testing. This perhaps hindered overall usability, but was necessary given the time constraints.

8.3 Software Design

8.3.1 Positives

The software design of this application benefited from the iterative development. In the early iterations, a focus on test driven development and code documentation was key in ensuring the main functionalities of the software design were robust enough to efficiently add new functionalities in later iterations.

The development cycle forced reflection at each stage, adding structure with set goals such as requirements and concept renders, ensuring the main functionalities of original vision were not confused in what otherwise would have likely been an unfinished application if the waterfall model had been selected. The development of this in essence took the planning stage, seen at the start of the waterfall model of development, and carried it out on a micro scale at the start of every iteration.

8.3.2 Negatives

The time scale between iterations had a serious effect on maintaining the good software development practices listed above. While decent reflection and code documentation remained, practices such as test driven development and good object orientated design were put aside in favour of having something viable to test with users.

The design of what resulted in the The Leaf Problem was caused by the iterative method. In order to rush the application towards user testing, the design of this was selected on what would be easiest to implement within the given timescale. What was intended to be temporary, resulted in being permanent as no more time could be allocated to replacing it as it was no longer viewed as a "core feature", but as a lower priority "usability feature".

8.4 Software Usability

8.4.1 Positives

Usability was undoubtedly improved by the iterations experienced in this project. The most obvious example would be the the solution to "The Root Problem" in Iteration 2. Here we see the feedback experienced at the user testing stage of Iteration 1 being fed into the requirements for the next iteration and both efficiency and satisfaction as regards to usability being increased at the next iteration's testing.

8.4.2 Negatives

A major effect the iterative development model had on usability other than the fact features which would have increased usability were cut was that the main functionality of the application had to be prioritised over user experience. This resulted in user interfaces with usability problems remaining throughout the whole development simply due to functionality taking priority. There is nothing other than increased time which could solve this however as a waterfall model equivalent project would likely have created a highly developed user interface that could not do anything as it was lacking core functionalities.

8.5 Software Testing

8.5.1 Positives

My testing methodology was extremely good at forming the basis for the requirements of the next iterations. I was able to receive constructive feedback, gauging what users did and did not like about the application and addressing it within the next set of requirements. It was also useful in understanding how the application was progressing across each iteration as criticisms became less centred around functionality and more about usability - A definite success of the iterative development.

8.5.2 Negatives

In hindsight I would have liked to have done a System Usability Scale questionnaire for every iteration of testing. This, and considering factors like the time and number of interactions[16] to complete the task would have given me quantitive feedback over what was in essence just qualitative feedback.

My largest problem with the iterative process however at the testing stage was that the application was not tested at two iterations with full functionalities (e.g. the photo timeline). This led to problems with respect to feedback not being aimed at the overall photo organisation aspect of this application, but at the irrelevant genealogical haves and have nots of the family tree element.

8.6 Evaluation

This application would with out a doubt have benefitted from a longer first iteration in which the photo functionalities were fully implemented before user testing. While the Second Iteration was extremely successful in addressing the feedback from the first iteration, only speculative feedback was given at these two iterations as regards to the defining photo organisational feature of this application. It was not until the third iteration, where participants were able to see the full use case of the application implemented in which I was able to get useful feedback with respect to the overall functionality.

Had some of the “Future Requirements” in the previous chapter been implemented, I do however believe that this would have addressed the above problem. Allocating a whole iteration to each of the two modules functionality and then an iteration given solely to usability features on each of the modules works extremely well in developing an application for the end user. It is unfortunate I was unable to carry out a fourth iteration to see the happen, but what was achieved, the completion of the use case with the core functionality, I am happy with.

8.7 Conclusion

I believe what has been created serves as a tool for visualising one's family history as a story through the years. The iterative design process has shaped this application in a way the waterfall model never could or would have finished. Usability, requirements and software design have all been refined to create a completed, unique piece of software for the end user in the given timescale.

Bibliography

[1] Michael Inglis. Managing and preserving personal photo collections for future generations. http://www.mingles.co/folio/urp, December 2014. (Accessed on 25/03/2016).

[2] Adel Alshamrani and Abdullah Bahattab. A comparison between three sdlc models waterfall model, spiral model, and incremental/iterative model. https://www.academia.edu/10793943/A_Comparison_Between_Three_SDLC_Models_Waterfall_Model_Spiral_Model_and_Incremental_Iterative_Model, January 2015. (Accessed on 25/03/2016).

[3] Apple. Photos framework. https://developer.apple.com/library/ios/documentation/Photos/Reference/Photos_Framework/, . (Accessed on 25/03/2016).

[4] Iso 25010. http://www.iso.org/iso/catalogue_detail.htm?csnumber=35733. (Accessed on 25/03/2016).

[5] Wikipedia. Breadth-first search. https://en.wikipedia.org/wiki/Breadth-first_search, . (Accessed on 25/03/2016).

[6] Wikipedia. Depth-first search. https://en.wikipedia.org/wiki/Depth-first_search, . (Accessed on 25/03/2016).

[7] D3.js. https://d3js.org/. (Accessed on 25/03/2016).

[8] The New England Historic Genealogical Society. American ancestors. http://www.americanancestors.org/. (Accessed on 25/03/2016).

[9] Wookiepedia. Skywalker family tree. http://starwars.wikia.com/wiki/Skywalker_(family)#Family_tree. (Accessed on 25/03/2016).

[10] Wikipedia. Pedigree chart. https://en.wikipedia.org/wiki/Pedigree_chart, . (Accessed on 25/03/2016).

[11] justincy. Github. d3-pedigree-examples. https://github.com/justincy/d3-pedigree-examples. (Accessed on 25/03/2016).

[12] Wikipedia. Genogram. https://en.wikipedia.org/wiki/Genogram, . (Accessed on 25/03/2016).

[13] Apple. Example app using photos framework. https://developer.apple.com/library/ios/samplecode/UsingPhotosFramework/Introduction/Intro.html, . (Accessed on 25/03/2016).

[14] ooper shlab. Github. samplephotosapp-swift. https://github.com/ooper-shlab/SamplePhotosApp-Swift. (Accessed on 25/03/2016).

[15] Usability.gov. System usability scale questionnaire. http://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html. (Accessed on 25/03/2016).

[16] Jeff Sauro. 10 benchmarks for user experience metrics. http://www.measuringu.com/blog/ux-benchmarks.php, October 2012. (Accessed on 25/03/2016).

Appendix A

Additional Materials

A.1 Testing Files

All of the following documents are supplied with the online submission material:

- Interview Audio (contains audio files of interviews)
- Testing - Information Sheet
- Testing - Participant Consent
- Testing - Question Sheet
- Testing - Interview Transcripts

A.2 Notes on Appendix

The length of the below Appendix files were too long to include in the printed document.

In the digital submission, these files can be accessed within the folder titled "Appendix Files".