Schedula

A University/College Course Scheduling Application for the iOS Platform

by

Cole Dorma

100965295

November 2017

Supervised by Dr. Anthony White

School of Computer Science

Carleton University

Honours Thesis (COMP 4905)

Abstract

Every year, students in post-secondary education have to decide the courses they need/want to take. This decision will be based off of their degree specifications, outside priorities, and preferences. After this decision, they then will need to build their course schedules depending on all of the possible time slots and sections for each course. This can be very stressful, difficult, and time consuming. Carleton University is an example of one of the many universities/colleges that does not have a tool that produces all of the possible course schedules for you. Schedula is an application that allows for a student to select all of the courses they want/need to take, and then select a preferred time that they would rather attend classes (e.g. morning, afternoon, night), select specific times during the week that they are not able to attend classes (e.g. 2:30 PM - 5:30 PM, hockey practice), and select full days of the week that they are not able to attend classes. Based off of the courses that the student has selected and their selected preferences, Schedula will generate all of the course schedules that are possible.

Acknowledgements

I would like to thank Dr. Anthony White for supervising this project and believing in the idea of Schedula. He gave me the opportunity and freedom to build Schedula how I first envisioned it. I hope Schedula will be of use to many students and himself at Carleton, and hopefully along with many other universities/colleges as well.

Table of Contents

List of Figures

Introduction and Background

Introduction

Every single year, post-secondary students are having to decide on the courses they want to take and the courses they need to take based off of their degree specification and requirements, and their personal preferences. When deciding on these courses, there is multiple different sections for a specific course, along with multiple different time slots for each section. This gives a lot of options and freedom for a student, but then comes the daunting task of building their course schedule with all of the courses they have chosen. For every student, building a course schedule can be very time consuming, stressful, and difficult. The majority of students have to write or type all of the possible sections and time slots for each course, then write or type all of their outside priorities, then compare all of these inputs to see if any course combinations happen to work out in the end. If there is not any possible combinations, the student then needs to go back to their initial step, decide on another course, or several different courses to try and find a course schedule that works for them. These negative traits that come with building a course schedule are multiplied by magnitudes when the students' school does not provide any sort of tool to help with this. Carleton University is one of those schools that does not provide a sufficient course scheduling tool, in which the majority of students spends hours upon hours building their course schedules. Out of a questionnaire that I gave to 18 Ontario universities, only 4 universities confirmed that they had a tool that helped with course scheduling (refer to the Questionnaire section in the Appendix for these results). Out of those 4 universities, I reached out to 5 students at those particular universities and asked them to give feedback on that tool. The majority of those students felt that the tool provided by their university was not sufficient and they still reverted to the original way of building their course schedules by writing or typing all of their course sections and time slots and comparing. The general reasons for the insufficient existing tools given by the students were that it didn't allow them to enter in all of their courses and generate all of the possible schedules based off of these courses, it didn't allow them to enter in their outside priorities, it didn't allow them to generate schedule possibilities for more than one course, and that it didn't generate schedules for them at all.

The main thing that Carleton University, along with many other universities, is missing, is a tool that allows students to select the courses that they want to take, input their outside priorities and preferences, and the tool generates all of the possible course schedules for them within seconds. Allowing them to view all of these schedules visually and pick the one that appeals most to them. The current tools that I have been able to personally use and do research into are Carleton University, and Trent University's tool. I was not given access to Wilfrid Laurier University's or Western University's tool. Using Carleton's tool for the past 4 years now, it allows you to select courses to put on your course timetable and it will notify you after adding the course, if it conflicts with any other courses. From my short-term usage of Trent's tool, I gathered that it allows you to select courses and it will generate possible course schedules, only for the exact section and time slot you have picked for each course, resulting in very minimal help for students building their schedules. Both of these tools do not appear to be able to take in personal preferences and outside priorities, nor do they appear to be able to generate all of the possible course schedules based on the overall courses selected (not the specific section and time slot), the outside priorities inputted, and the personal preferences inputted.

This project idea was first thought of during my second year when I was building my course schedule for my third year with some friends and we asked each other, "why is this so time consuming, stressful, and difficult?" This project was based off of that one question that we all asked each other. Throughout this honours project, the first section consists of a description of the issue and the rationale, along with the background information in regards to the technology and methods used in the project. The following section discusses the goals and objectives of this project. The next section discusses the application design and the challenges at each aspect of the application design. The last section consists of the results and the conclusion of the project.

Background

Xcode

Xcode is software that can only be installed on Mac OS. It provides an efficient and effective environment to build iOS applications. Xcode's main language support is for Swift, Objective-C, C, and C++, but a variety of language are supported such as Ruby, Python, AppleScript, and much more [Anon., 2017]. Xcode includes the iOS SDK tools in order for you to use them in your iOS applications. It also has a visual interface builder within it to develop the user interface (UI) of your application and to be able to visually work with some aspects of your iOS application rather than working with some of the visual aspects programmatically only, like developing Android application in Eclipse. On top of this, Xcode also has virtual device simulators built in that allow you to test your iOS application directly on a virtual device, rather than testing it only on a real iOS device plugged into your Mac OS system. Xcode provides a variety of virtual devices as well. At the current time of writing this, it provides an iPhone 4S, iPhone SE, iPhone 7, iPhone 7 Plus, iPad Pro 9.7", iPad Pro 10.5", iPad Pro 12.9". This allows for much quicker testing, with the same accuracy as testing on a real iOS device.

There are many different types of software that allow you to build applications for the iOS platform but a lot of them mainly rely on Xcode's tools and newer implementations to work, hence they are usually a few version behind in regards to new features and development tools. These other different types of software in which you can develop iOS applications also do not usually have the full package of testing, visual programming, and simulators like Xcode, resulting in the developer having to spend more time to migrate multiple different softwares together as one fully functional piece of software equivalent to Xcode, rather than building an iOS application.

In the end, I chose to use Xcode for this project due to it is competitive edge in iOS application development. Xcode is developed and maintained by Apple and has been constantly updated with new features and tools every few months for the past 2 years [Anon., 2017]. It has advanced built-in testing, a visual programming system, and it also has iOS device simulators that allow you to visually test your application quicker and just as accurate as a real iOS device. It worked best for my project, as my project was very visually reliant, and also it has the iOS SDK, allowing me to use all of the most updated and advanced tools for my project.

Swift

Swift is a programming language used to develop applications for Apple product software like macOS, iOS, watchOS and tvOS. Swift is a very new language, in which it was first released in late 2014 [Anon., 2017]. It is a very powerful object-oriented programming language that was built off of the research of many other languages, some of those being Objective-C, C++ and Java. Swift has the ability to still be implemented in a project alongside Objective-C code as well. It is a simple programming language that abstracts the syntax and data types, as well as manages memory automatically for you. Swift also has great error and warning catching, with suggestions to fix those issues as well. It was ultimately created to bring more ease to the programmer and allow for more programming, with less error and syntax fixing.

The main alternative to Swift for iOS application development is Objective-C, followed by C++. Objective-C is still used alongside Swift for iOS development, which is even advertised by Apple themselves. Swift has only recently taken the stage from Objective-C, where Objective-C has been the main language for iOS development since the late 1980's. Everything in the iOS SDK was built in Objective-C [Premaratne, 2016].

Considering the main alternative, Objective-C, I chose to build this project in Swift. I did this mainly because I have been learning and developing iOS applications with Swift since 2015, from which I have grown to be proficient in Swift. Swift worked best with my project because it provides many tools with the iOS SDK that allowed me to develop the application in Xcode, with Xcode providing me with visual programming, testing, and device simulation that my project was heavily reliant on. Also, I have not ever learnt, or developed any type of software or application in Objective-C before, which is another main reason I chose Swift.

XMLParser

A Parser is a tool that parses through a document line by line, and notifies the developer of the information that they have specified, quickly. The XMLParser is a built-in tool in Swift that parses an XML document, and notifies a delegate, of the information it is parsing, in which this delegate can do something with that information.

For my project, all of the courses, course sections, time slots, and days needed to be parsed and stored on the device, quickly. This stored course information was then being used for when a user selected a course. The fastest and best way that I could achieve this was with the built-in XMLParser that Swift provides. I used this XMLParser in which it parses each line of the courses information that is stored online as an XML document, and then stores that course information locally on the device in an array for later use by the algorithm.

Goals and Objectives

The motivation for this project came from my own personal, along with many other post-secondary students' struggles of building course schedules based off of the courses that we decided to take. Each and every year it takes me, along with every other student I have spoken to, multiple hours to finish figuring out a course schedule that works best. From my research through questionnaires with Ontario universities, explained in the first paragraph in the Introduction section above, there is not a tool that allows for a student to select the courses they would like to take, enter in specific times and full days during the week they are not available to attend class, enter in preferred times during the day to attend class, and the tool generates all of the possible course schedules for them in milliseconds. Our entire world is moving to the mobile platform, therefore this tool needed to be built on the mobile platform. This tool needed to be simple to use, in regards to being able to select courses easily with a text box and a button, and select preferences easily with toggle buttons and autofilled sliders, as it is not something a student would be seeking entertainment from, like a mobile game. Some user interface design patterns that would be appropriate to achieve this would be using navigational tabs, modals, input prompts, event calendars, completeness meters, and pagination. Our society is constantly moving at a very fast pace, in which not many people have, or can find a significant amount time for tedious tasks like building your course schedule. Therefore, this tool would also need to be very efficient, responsive, and quick throughout the entire system, meaning that when the user would be tracking from page to page, there would be no delays and would have smooth transitions, and when the user clicked on a button or moved an autofilled slider, the slider would respond immediately without any delays, and when the user generated their schedules, all of the possible schedules would be generated within seconds of clicking a button. The main challenge for this tool is to ensure that it is simple, efficient, and accurate to replace the old-fashioned pen and paper method of building a course schedule.

The motivation to build such a tool can only be successfully satisfied if the following objectives are completed:

1. Design and implement an iOS application that allows for students to simply select the courses they want to take, specific times during specific days they are not available to attend class, specific full days they are not able to attend class, and a preferred time during the day to attend class. On top of all of this, display all of the possible course schedules according to the students' selections.
2. Design and implement an algorithm that can take in the students' courses, outside priorities, and preferences as inputs and output all of the possible course schedules in a very quick, reliable, accurate and efficient manner. With this quick and efficient manner only being within the range of milliseconds.

Application Design

The project was developed in three separate parts and then merged all together; these parts were:

1. **Front End** - This part of the project included all of the different user interface pages, as well as all of the user interface components for each of those pages. This also consisted of all of the different screen size versions (iPhone 5/5S - 5 inch, iPhone 6/6S - 5 inch, iPhone 6 Plus/6S Plus - 5.5 inch, iPhone 7 /7S - 5 inch, iPhone 7 Plus/7S Plus - 5.5 inch) for each user interface page.
2. **Back End** - This part of the project included all of the back end code for the iOS application, all of the code for the underlying model, the back end code to connect the user interface components to the application, as well as handle all of the input to the different user interface components and perform some task with that input. This part also connected the front end to the algorithm.
3. **Algorithm** - This part of the project included the algorithm that took in the users' selected courses, specific times on specific days that they could not attend class, whole days that the user could not attend class, and their preferred time during the day to take class as an input. The algorithm then computed all of the possible course schedules based off of the inputs it received and outputted all of those possible schedules.

The application used the Model-View-Controller pattern. In the following parts, the structure of the Back End, consisting of the underlying model, and Front End together, as well as the Algorithm will be broken down in a descriptive manner.

Front End and Back End

The front end and the back end code of this project are made up of the the following corresponding main classes (Back End - Front End):

1. **ViewController** - The first user interface page that the user is prompted with. The home page, which allows the user to select their university and start.

2. **CourseTabController** - The second user interface page, the course selection page, which allows for the user to select their courses for the Fall and Winter semesters.
3. **SecondViewController** - The third user interface page, the preferences pages, that allows for the user to select a preferred time of day to attend classes, specific full days that they are not able to attend class, and specific times during specific days that they are not able to attend class.
4. **CalendarCollectionViewController** - The fourth user interface page, the course schedules page, which displays all of the possible course schedules that were generated from the algorithm.

The following use-cases for this application will use these use-case ID's:

1 Allows users to generate up to ten possible timetables

1.1 User selects semester

1.2 User enters course codes in search and bar and clicks add

1.3 User selects time periods they have strict commitments

1.4 User selects preferable time of classes (morning, afternoon, or evening)

1.5 User selects preferable days of the week off

1.6 User clicks to generate possible schedules

1.7 User selects most ideal schedule and course registration numbers are displayed

1) Student A is currently working full time during the day and has decided to change career paths by getting a degree. She plans on studying part time, however would like to keep her current job while working towards a degree. This means taking evening classes. At 1.1, she selects the Winter semester and enters in two courses at 1.2. She works 8:30-4:30 Monday to Thursday and 8:30-2:30 on Friday so at 1.3 she enters these times as strict commitments when she cannot have class. Her children also have hockey practice on a Monday at 8-10pm, so she adds this to 1.3. Since her class schedule is purely built around her work schedule, she can simply leave 1.4 and 1.5 empty. Based on her course selections, and number of course sections and tutorials/labs, at 1.7 she is now able to choose from a maximum of 10 mock schedules. Once

she chooses the most ideal timetable, the CRNs are displayed to be easily entered into Carleton Central.

With student A's already incredibly busy life, Schedula begins her journey at Carleton simply. The selection of the very best timetables available will help her to manage her work life, family life, and school life as efficiently as possible, during the semester. Ultimately, student A does not have time to waste in both making her schedule and during the semester, and Schedula is the perfect solution.

2) Student B is entering his second year at Carleton. He will be playing for the varsity soccer team as well as studying this Fall semester, so he selects the semester at 1.1 and enters his courses at 1.2. He has soccer practice Monday and Tuesday evening, and has a weekly game every Thursday evening. These commitments cannot be altered so he enters in the exact times at 1.3. Having practices and matches in the evening, he has a daily diet plan that requires time to rest before playing. So, at 1.4 he chooses morning and afternoon classes as a preference. His Thursday matches can either be home or away. In the case they happen to be away matches, he won't be returning home till very early in the morning. After a very busy week, ideally he would like to have Fridays off and enters this in at 1.5. Based on his course selections, and number of course sections and tutorials/labs, at 1.7 he is now able to choose from a maximum of 10 mock schedules. Once he chooses the most ideal timetable, the CRNs are displayed to be easily entered into Carleton Central.

Student B has a very particular schedule with his involvement in the soccer team. Manually finding the best schedule would not only take a lot of time, but also a lot of patience! His use of 1.4 and 1.5 allows him to test preferences with no time lost if it proves to be impossible regardless based on his course selections. This is why Schedula is perfect for him.

3) Student C is entering her first year at Carleton. She is new to the whole idea of making her schedule, as she had a lot of guidance during high school. She knows what classes she wants to take so enters the semester and the courses at 1.1 and 1.2. She only has one commitment during the week so enters Wednesday from 8-10am at 1.3. She knows she might like to have

afternoon classes and maybe a day off, but other than that she is really indecisive. She wants to see possible timetables in front of her and then be able to select from a wide range. So, she enters both morning and afternoon classes at 1.4 and leaves 1.5 empty. Since she has selected few restrictions and preferences, the max number of schedules are generated, and at 1.7 she is able to choose the timetable that she prefers the most.

With the newness of university, student C wants options and choice. She doesn't want to make any rash decisions. The ease of Schedula allows her to take her time and see what variations her course selection has to offer. Schedula gives her the start to university that she needs!

The following is a use-case for the majority of the users of this application. This use-case being used for explaining the each and every aspect of the application. Once starting the application, the user will be prompted with the first user interface page, the welcome page. All of the input, drawing, and processing is programatically handled in the corresponding ViewController class for the welcome page

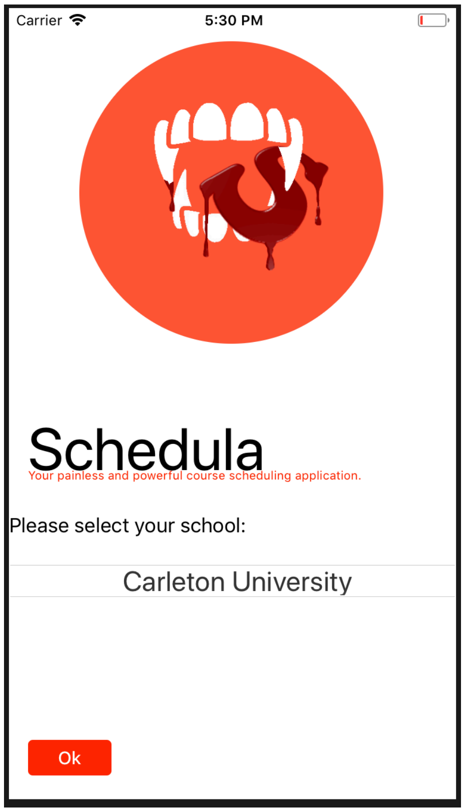

Carrier
5:30 PM
Schedula
Your painless and powerful course scheduling application.
Please select your school:
Carleton University
Ok

FIGURE 1 The first user interface page, the home page.

ViewController

The ViewController class is the class that allows for the user to select their specific university/college, from which, the appropriate courses are loaded locally into the application for later use in the application. The welcome page was motivated by the completeness meter user interface design pattern, as when the user selects their specific school, a loading bar will appear to show the status of the courses that have been loaded into the application until all of the courses have loaded into the application for their specific school. The only school that has provided their course data is Carleton University. During developing this application I came across the issue that course information isn't information to the public and is only accessible by a registered student at that specific school. Therefore, this is a limitation in the sense that this application is limited to only being able to give support to schools that decide to give their course information, for that specific year, to me for use in this application. On top of this, the course information would need to be obtained from each school that has decided to be supported by this application, at the beginning of every school year due to the fact that courses, course sections, course times, and course instructors do not remain consistent year to year. The user will first have to select their university/college from the dropdown menu before being able to proceed to the next step with the "Next" button located at the bottom right of FIGURE 1. Once the user selects their university/college from the dropdown menu, the ViewController process this selection as a String and activates the XMLParser to parse that selections' course information. The course information for each supported university/college (only Carleton University as of now) is stored as an XML document at an online link which directs you to the privately hosted XML document. The XMLParser then will parse all of the information that it is directed to, which will be the specific link that is being passed to the XMLParser based on the user's school selection. This archive of XML documents would be maintained by myself. This is a definite limitation of the application as storing all of this information in a weblink is tedious and inconvenient. It also takes roughly 22 seconds for the XMLParser to parse all of Carleton's course data as well, which is another negative aspect of using solution, as a database would eliminate this time down to the amount of time it takes to access the database. Ultimately, a database will be the only solution for this but due to the time constraint of this project, and my lack of database knowledge, an XMLParser and

XML document was a solution that allowed me to finish the project on time. The XML document is in the form of:

```
<Row ss:Index="53">
    <Cell><Data>201630</Data></Cell>
    <Cell><Data>30055</Data></Cell>
    <Cell><Data>ALDS</Data></Cell>
    <Cell><Data>4205</Data></Cell>
    <Cell><Data>A</Data></Cell>
    <Cell><Data>LEC</Data></Cell>
    <Cell><Data>Rodgers, Michael Patrick Hindley</Data></
     Cell>
    <Cell><Data>MW 0935-1125</Data></Cell>
</Row>
```

The form of this XML document isn't ideal, as this form forces position to imply semantics, but this XML document is what was provided to me by Carleton, in this exact format. The ideal format of this XML document would be each <Cell> tag containing the correct tag according to the information that is within that row, like the first <Data> tag being replaced with a <Year> tag. In order for the XML document to be in the previously stated ideal form, I would have had to go through each line of the entire document supplied to me by Carleton and change each of those specific tags to more appropriate tags. This would have taken upward of 20 hours, which is time that I didn't have to spare, especially for data entry. The XMLParser now intricately parses the XML document line-by-line, dissecting and pulling the data by first entering the <Row> tag, meaning that the XMLParser is now in a new course, then entering the <Cell> tag, quickly followed by the <Data> tag at each line. The XMLParser now pulls the text within the <Data> tags lines with the first line in the new course representing the year (201630), the second line representing the CRN code of the course (30055), the third and fourth line being the course code (ALDS 4205), the fifth line being the section of the course (A), the sixth line representing the whether the course is a lecture, lab, or tutorial (LEC), and the seventh line being the specific time and days that the course is held on (MW 0935-1125). For the information pulled, the XMLParser creates Course objects for each course it parses and stores these course objects in separate semester arrays that are then stored locally on the device. The Course object consists of a code and sections, in which the sections is an array of Section objects, which contain the ID, prof, crn, times, and subSec. These Section objects represent the different sections that are available for a course (e.g. COMP 1405A, COMP1405B). The subSec is an array of SubSection objects, which is a child of the Section class. These SubSection objects represent the tutorials and labs for a course section (e.g. COMP 1405A1, COMP 1405A2) that are mandatory apart from the main lectures of some courses. The user is now able to click the "Next" button, where these separate semester arrays are passed to the next user interface with Swift's built-in function to pass information from one UI to another UI, a segue, and this will bring the user to the next user interface page, the course selection page.

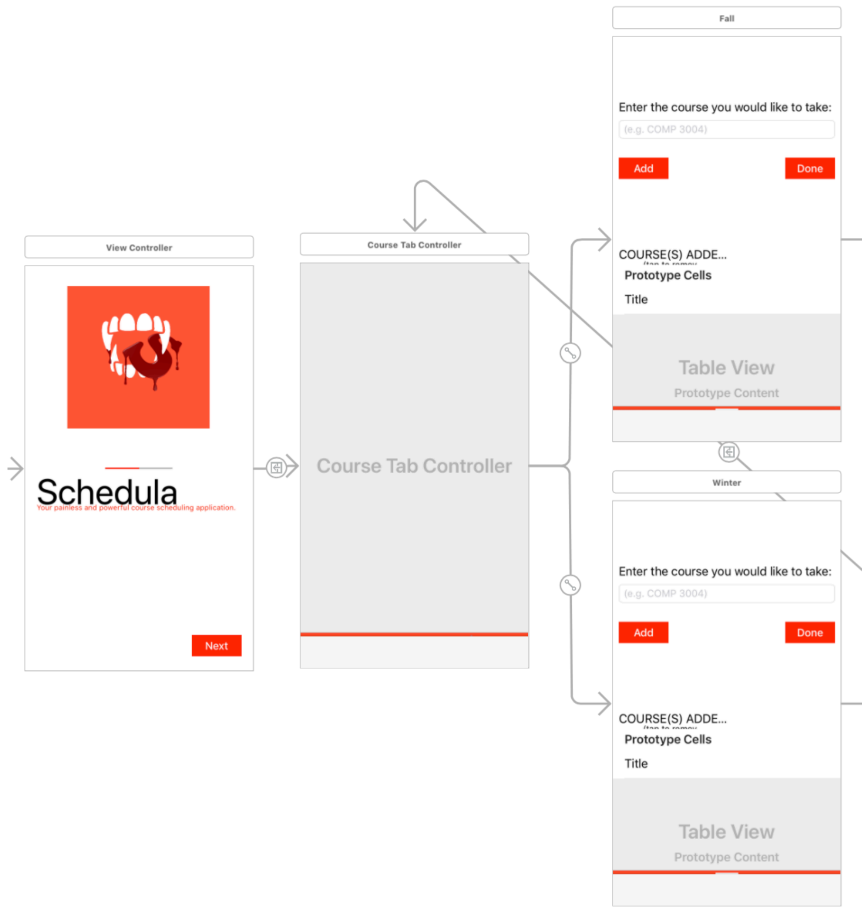


FIGURE 2 The segue movement between the home page and the course selection page.

All of the input and processing for the course selection page is handled by theCourseTabController class.

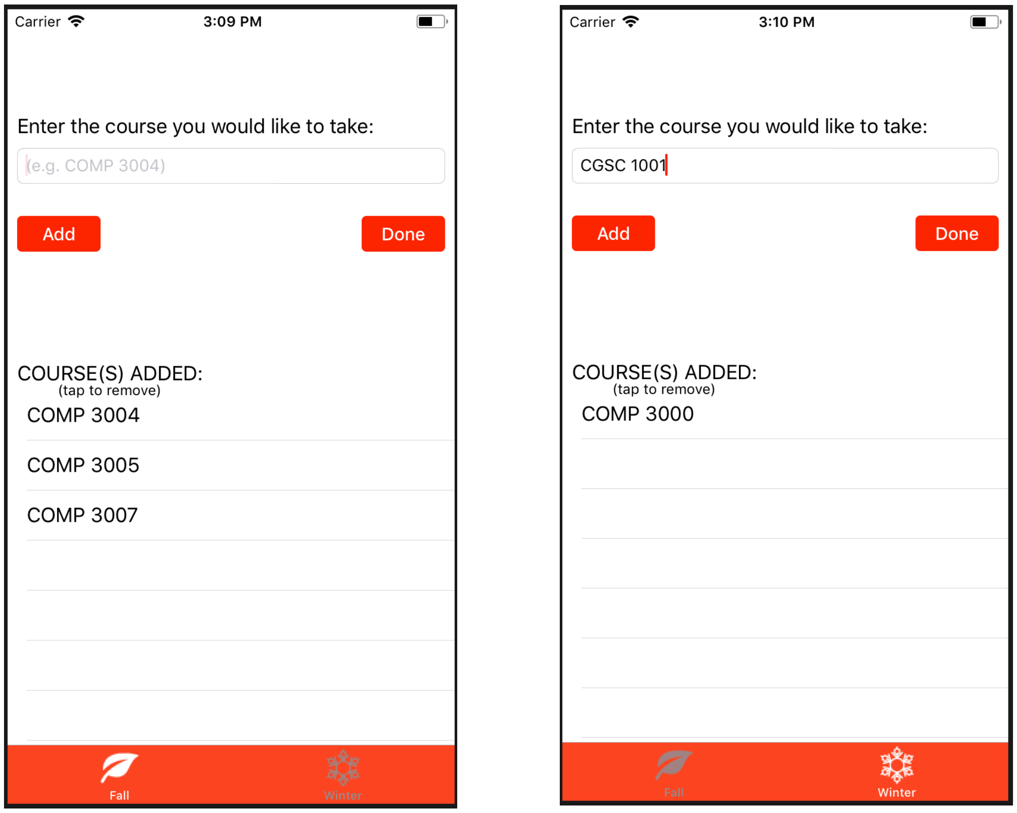
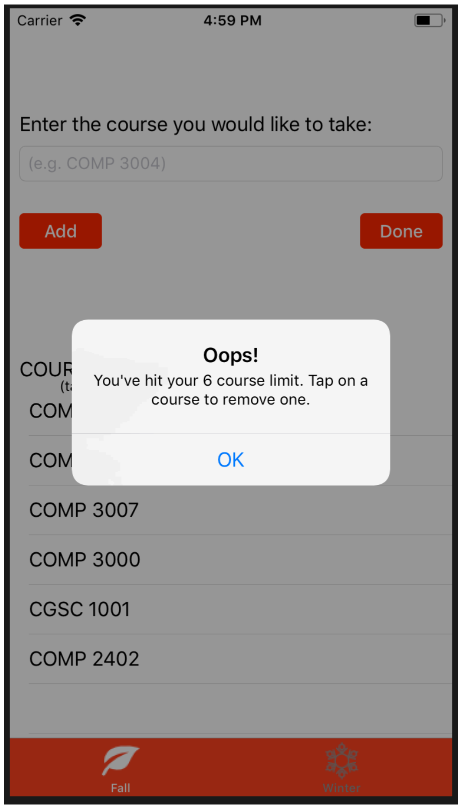


FIGURE 3 The second user interface page, the course selection page.

CourseTabController

The CourseTabController class is the class that manages the Fall and Winter semester tabs that you can see at the bottom of FIGURE 3. This user interface doesn't include any option for a Summer semester is due to the fact that Carleton University only provided me with the Fall and Winter semester courses. An additional Summer tab and controller could easily be implemented, as it follows the exact same template as the Fall and Winter semester tab and controller templates, if the Summer semester courses are provided by Carleton University, or any other school that provides their course information. This course selection page was motivated by the modal user interface design pattern, the navigational tab user interface design pattern, and the the input prompt user interface design pattern. As when the user enters in an incorrect format of courses, an error message will be prompted to the user (which can be seen in FIGURE 4) as a modal, when the user navigates to either the Fall or Winter it is controlled through navigational tabs, and the action of the user entering in their courses as an input prompt. It controls which display will be showing by which tab is selected; either it notifies the device to display the Fall user interface or the Winter user interface. The Fall user interface's Fall course semester array, which is passed by a segue from the CourseTabController class, course selection inputs, and the selected courses array are handled by the FallViewController class. The Winter user interface's Winter course semester array, which is passed by a segue from the CourseTabController class, course selection inputs, and the selected courses array are handled by the Winter ViewController class.

The user now is able to select their courses that they want to take for either the Fall or Winter semesters by typing the course name in the text box and clicking the "Add" button. Once a course is entered and the "Add" button is clicked, the course is found by the course name in either the Fall semester array or Winter semester array (depending on what tab the user is currently in) with the built-in "index" method in Swift. If the course exists in the correct semester array, and if adding this course does not exceed the maximum number of 6 courses that can be taken each semester, the course is added to the courseArray array. If the course cannot be added for one of the before stated reasons, this error is prompted to the user with an alert, using Swift's built-in UIAlertController and UIAlertAction classes.


Carrier 4:59 PM

Enter the course you would like to take:

(e.g. COMP 3004)

Add

Done

COUR

COM

COM

Oops!

You've hit your 6 course limit. Tap on a course to remove one.

OK

COMP 3007

COMP 3000

CGSC 1001

COMP 2402

FIGURE 4 Alert error message prompted to the user when an error has occurred with the current action attempting to be made.

After the course is added to the courseArray, it will appear in the “COURSE(S) ADDED:” table view object. The table view object is an object built-in to Swift as a TableViewUI, that allows for data to be displayed in the table view’s cells. In this projects case, the data being displayed to the user in the table view is the courses that the user has selected, which are stored in the courseArray array. Each cell in the tableView corresponds to an index starting at (0, 1, 2, 3, …. , n). These cells are populated with Swift’s built-in methods for the table view object corresponding to each index in the courseArray. If the user has added a course by mistake, or they have decided to change a course they have selected, they can simply tap on the course to delete it, which is indicated to the user underneath the “COURSE(S) ADDED:” text in FIGURE 4. This was implemented with one of Swift’s built in methods for table view objects, this method being the “didSelectRowAt” method.

```
override func tableView(_ tableView: UITableView, didSelectRowAt
indexPath: IndexPath){
```

This method was used by getting the index of the row that was tapped on by the user, which is pulled automatically from the method above, then by using this index to remove the element in the array of courses pertaining to the course that was tapped on by the user, which populates this particular table view. The table view was then reload, displaying to the user the new table view of courses after the specific course they tapped on was removed.

Once the user has added all of the courses they want to take for their selected semester, they can then click the “Done” button, then according to the selected semester (e.g. Fall or Winter), that selected semester’s class (e.g. FallViewController or WinterViewController) will pass the current selected courses array to the next user interface page. That user interface page being the third page, the preferences page. The preferences page’s input and processing is all handled by the SecondViewController class.

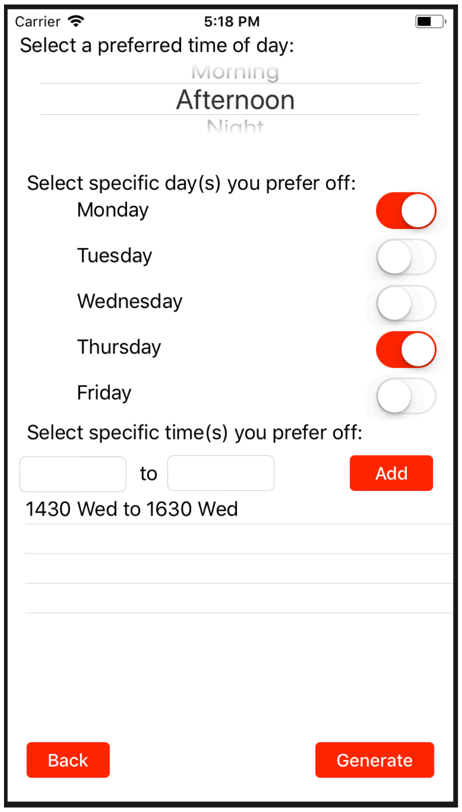

Carrier
5:18 PM
Select a preferred time of day:
Morning
Afternoon
Night
Select specific day(s) you prefer off:
Monday
Tuesday
Wednesday
Thursday
Friday
Select specific time(s) you prefer off:
to
Add
1430 Wed to 1630 Wed
Back
Generate

FIGURE 5 The third user interface page, the preferences page.

SecondViewController

The SecondViewController class manages and handles all of the user’s personal preferences for attending class. This preferences page takes motivation from the event calendar user interface design pattern, in the sense that when the user has added a specific time, it will appear in a list. It also takes motivation from the carousel user interface design pattern, in which the preferred time of day setting is displayed in a carousel type view, as well as when the user is selecting their specific times. It specifically handles the user's preferred time of day that they wish to attend classes, the specific day/days that the user prefers to not attend class, and specific times for specific days that the student is unable to attend classes.

The preferred time of day preference is handled by a picker view which is an object that is built-in to Swift that allows for data to be displayed in an infinity-like scroll wheel, which can be seen just below the "Select a preferred time of day:" text at the top of the screen in FIGURE 5. This preferred time of day picker view was implemented in the SecondViewController class as the built-in UIPickerView object. This picker view was then populated with an array of strings correlating to the options being displayed in the infinity-like scroll wheel: morning, afternoon, and night. This array of strings was then attached to the picker view with Swift's built-in functions for the picker view object.

The specific days that the user prefers not to attend classes are each handled by a switch which is a object built-in to Swift called a UISwitch, which can be seen in the middle of the screen in FIGURE 5, just underneath the text labelled "Select specific day(s) you prefer off:". Each of these switches correspond to a specific day in the week, and each day of the week corresponds to a boolean variable. When a switch is turned on, it will appear red, or off, it will appear white, it will respectively turn that specific day's boolean to either true or false.

The preference for specific times during specific days that the user is unable to attend class was difficult to implement, as it needed to contain a quick and efficient way of entering in these times for specific days, and also a way of displaying these times for specific days that the user has already entered, whilst fitting all of this into the preference page's user interface. After consulting many different options for this section of the preferences page, it was finally handled by a collection of Swift's built-in objects; a picker view and a table view. When the user taps on an empty text box located just below the “Select specific time(s) you prefer off:” text near the middle of the screen in FIGURE 5, a picker view, which was explained in the second paragraph of this section, animates upwards from the bottom of the screen.

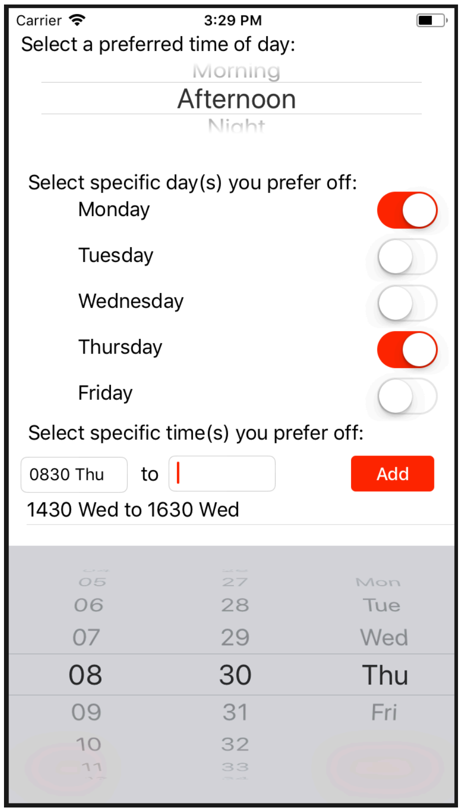


FIGURE 6 The picker view after the user clicks on an empty text box.

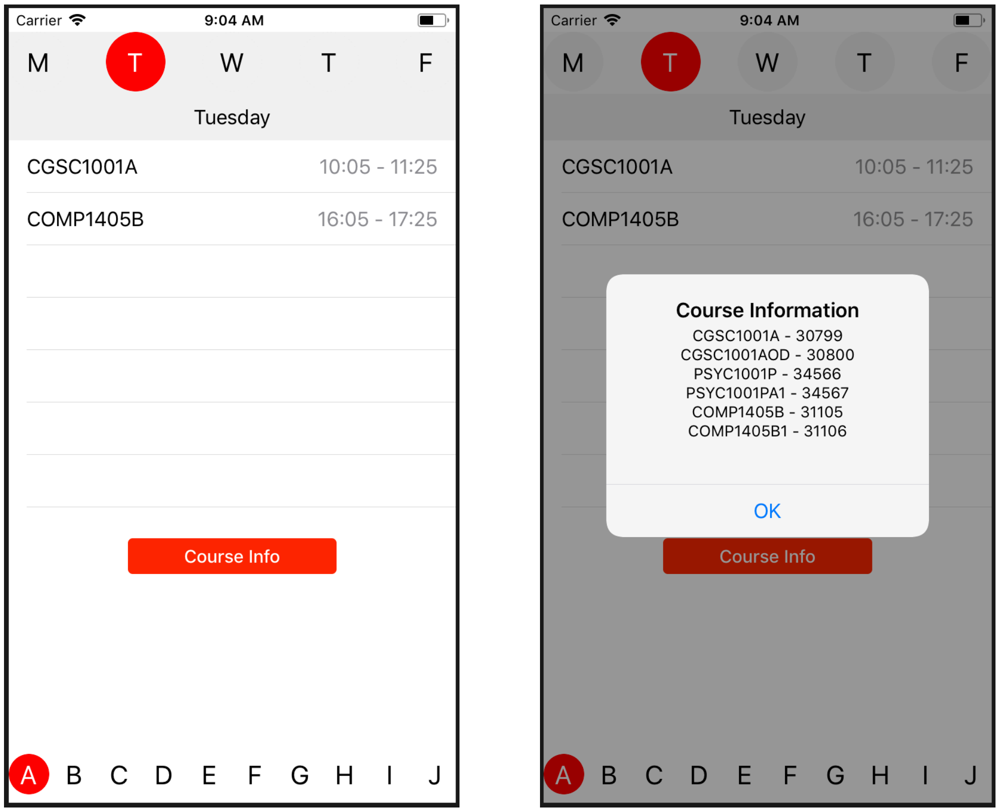
The picker view that is revealed is created the exact same way as the picker view that was explained above for the preferred time of day preference. It is just slightly modified to appear from the bottom of the screen after clicking on a text box, rather than being a static picker view object in the user interface, and has a different tag associated to it to differentiate between the two picker view objects. The appearance and animation of this picker view was done by setting the inputView property of the empty text boxes to this preferences picker view object.

```
begTime.inputView = preferredTimePicker
endTime.inputView = preferredTimePicker
```

Once the user picks their specific beginning time and ending time, and selects the “Add” button, that specific time is added to an array of Commitment objects (refer to FIGURE 11 and 12 in the

Appendix section), which are objects that consist of the type of commitment it is, the term that the commitment is in, which is related to whether the user has selected schedule generation for either the Fall or Winter Semester, and the time that the commitment is, as a SubSection object (refer to FIGURE 11 and 12 in the Appendix section), explained above in the first paragraph of the ViewController section. Once this Commitment object has been added to the array, a table view object is used to display these specific times that the user has added to the array. The table view object used here was created and used exactly as it was explained in the second paragraph of the CourseTabController section.

Once all of the user's personal preferences for attending class has been entered, all of this information, which is stored in 3 separate arrays: an array for the preferred time of the day, an array for the specific days the user preferred to not attend class, and an array for the specific times the user cannot attend class, is passed with Swift's built-in function to pass information from one user interface to another user interface, a segue, which was explained in the first paragraph of the ViewController section. This segue is initiated once the user clicks the "Generate" button, which can be seen at the bottom right of the screen in FIGURE 5. At any time during the user being on the time preferences page, they have the option of clicking the "Back" button, which can be seen at the bottom left of the screen in FIGURE 5, which will return the user back to the previous user interface page with a segue function, the course selection page. If the user clicks on the "Generate" button, the segue function associated to it passes the information explained at the beginning of this paragraph to the algorithm, and transitions the user to the fourth user interface page, the course schedules page.



CGSC1001A 10:05 - 11:25
COMP1405B 16:05 - 17:25

Course Information
CGSC1001A - 30799
CGSC1001AOD - 30800
PSYC1001P - 34566
PSYC1001PA1 - 34567
COMP1405B - 31105
COMP1405B1 - 31106

FIGURE 7 The fourth user interface page, the course schedules page.

CalendarCollectionViewController

The CalendarCollectionViewController class manages and displays all of the generated schedules that are outputted from the algorithm based on the users selected courses and personal preferences. This course schedules page takes motivation from the event calendar user interface design pattern, in which it displays the courses for the selected day in a list. It also takes motivation from the pagination user interface design pattern, in which it allows the users to select different schedules at the bottom of the screen and different days for that selected schedule at the top of the screen. This user interface page was the most difficult page to design as Swift did not have any type of built-in calendar view at the time of developing this project. This page was built with Swift's built-in collection view's, label, and table view objects. The inspiration for displaying the generated schedules in this way came from the iOS Calendar application.

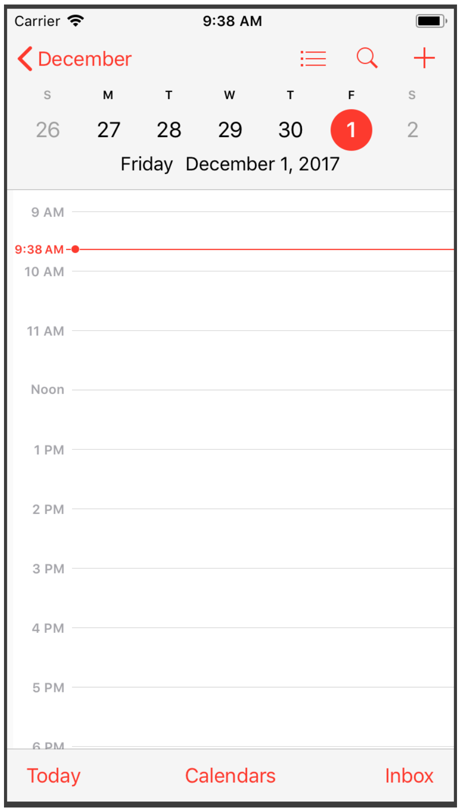


FIGURE 8 The iOS Calendar application.

The inspiration was directly influenced by the top section seen in FIGURE 8, that separated each day of the week, while allowing for the user to select a specific day, which was then highlighted with a red-filled circle. This exact styling can be seen at the top and bottom of the left screen in FIGURE 7. The top section represents each day of the week, while the bottom section represents all of the generated possible schedules. The course schedule page also took inspiration from displaying that highlighted day's information, which, in the project's case, was the course information for the selected schedule at the bottom of the screen.

The specific days of the week and all of the generated possible schedules, which can be seen at the top and bottom of the left screen in FIGURE 7, were implemented with two different collection view's that were differentiated from each other by their tag property. The collection view object in Swift is populated by cells. The collection view representing the days of the week was populated by an array of strings, where those strings were the first letters of each day of the week. The collection view representing the generated possible schedules was populated by how many schedules the LinkedList was holding of the generated schedules, with each of those schedules labelled with an array of strings containing the alphabet up to and including "J", to represent a maximum of 10 possible schedule options for the user, if there happens to be that much. The 10 possible schedules was chosen because I felt that 10 possible schedules was enough options for the user to find a schedule that works best for them. The selection ability was implemented with Swift's built-in function for seeing if an item was selected in a collection view, then distinguishing between the two collection view's in the user interface by their specific tags.

```
func collectionView(_ collectionView: UICollectionView, didSelectItemAt
indexPath: IndexPath)
```

Once distinguishing which collection view the selection was in, that collection view's cell that was selected would be changed to red with it's background property, and the radius property was increased for the circular appearance. Depending on which collection view the selection was in, this would trigger different events. If the user selected a different day, that day's full name would be displayed in the label right underneath the collection view's day cells, and that day's courses would be displayed in the table view object used in the middle of the screen, which can be seen on the left screen in FIGURE 7 right underneath the day label. If the user selected a different schedule, that schedule would now be loaded into each specific day of the week of courses that needed to be displayed in the table view.

Each generated schedule that is being displayed is stored in a LinkedList of Schedule objects (refer to FIGURE 11 and 12 in the Appendix section) that were generated from the algorithm. Then depending on the schedule that is selected, the corresponding days in each Schedule object are stored in their own separate string arrays consisting of the course name and the course time. These corresponding day arrays of strings are then displayed in the table view by populating the table view's cells with the strings in the array of the day that is selected.

After the user has viewed some, or all of the schedules that have been generated for them from the algorithm, and they have decided on a specific schedule that they would like to continue with, they can click the "Course Info" button, which will then display all of the specific course registration information to the user for the schedule they have selected, in an alert, which was explained above in the second paragraph of the CourseTabController section.

Model

The underlying model of this project consisted of the following classes: Timeslot, Section, Subsection, Commitment, Course, Node, LinkedList, Schedule. These classes were used to create, store, manage, access, manipulate and pass information throughout the application between the user interface and the controller classes. The exact instances of these actions were explained throughout this section as they were needed by either a controller class or the user interface. Please refer to FIGURE 11 and 12 in the Appendix for a component diagram showing the flow and uses between the model classes and the controller classes.

Algorithm

The algorithm used many different custom-defined objects, in which the following table outlines all of them for reference.

Object Name	Object Variable: Type
TimeSlot	- dayOfWeek: String - startHour: Int - startMinute: Int - endHour: Int - endMinute: Int - termMonth: Int - termYear: Int
SubSection (child of Section)	- id: String - prof: String - crn: Int - term: Int - time: String - subSec: [SubSection]

Object Name	Object Variable: Type
Section (parent of SubSection)	- id: String - prof: String - crn: Int - term: Int - time: String - subSec: [SubSection] - times: [TimeSlot]
Commitment	- type: String - term: Int - time: String - times: SubSection
Course	- code: String - sections: [Section]
Schedule	- sections: [Section] - size: Int

FIGURE 9 Table of custom objects used in the algorithm.

The algorithm of this project is made up of the the following corresponding classes:

1. **MyComparator** - the class that is a helper class to the algorithm that sorts all of the algorithm's generated schedules by the time periods between courses.
2. **ScheduleGenerator** - the class that is the algorithm and builds all of the possible schedules based on the user's inputted preferences and courses.

The following is an in-depth breakdown of how the MyComparator class and the ScheduleGenerator class work, also how they work together to play the role as the algorithm and the sorter for the algorithm.

MyComparator

The MyComparator class is the class that handles all of the comparisons between TimeSlot objects to optimize the possible schedules for the user by displaying the schedules with the least amount of time difference between the TimeSlots. The variables that make up these TimeSlot objects can be seen in FIGURE 9. The TimeSlot objects used in this class represent the specific courses selected by the user and all of the personal preferences selected by the user on the preferences page. The TimeSlot object was chosen to be used for this class to allow for the comparisons to be made more easily with both the course, and the personal preferences being represented with the same object. Since some of the preferences were not exact time slots like the courses, these were accommodated for by modifying some of the information that was entered into the TimeSlot object to represents these preferences. If the user inputted an entire day that they would not be able to attend class, the associated TimeSlot object created for the preference would be from 00:00 until 23:59, instead of the a specific time, like a course, to allow for, simplicity and comparisons between like objects.

This class consists of two main methods, the compute method and the compare method. The compute method takes in the sections, which represent the courses and preferences in a possible schedule, a minimum counter and a maximum counter to calculate the time spread between the sections, and the specific day of the week to represent the day that the time spread compute method is being used for. It will then calculate the amount of time spread there is between each section for each section that is in the specified day, and output that time spread. The compare method takes in two Schedule objects, in which the variables that create this object can be seen in FIGURE 9. It then will use the compute method to calculate the time spread between each section for each specific day of the week. With each time spread being computed for each day of the week, the differences are being added for each of the two different Schedules being compared, until the end of the week, in which these differences are finally compared. The Schedule object with the lower difference throughout the week, meaning that the time spread between the users selected courses and personal preferences is the smaller number between the two Schedule objects, is then outputted.

ScheduleGenerator

The ScheduleGenerator class is the class that contributes to the overall algorithm the most. It takes in all of the courses that the user has entered as Course objects, the full days and specific times during the day that the user is not able to attend class as Commitment objects, and Strings of the user's preferred time of the day to attend courses (e.g. morning, afternoon, or night). The created Course object and Commitment object specifications can be seen in FIGURE 9. From these inputs, this class builds all of the possible schedules and stores them in a LinkedList of Schedule objects, which is then outputted and displayed on the fourth user interface page, the course schedules page shown in FIGURE 7. The LinkedList data structure was implemented from scratch, as Swift at the time of developing this project, did not have a built-in implementation of a LinkedList. It was chosen to store the Schedule objects for many reasons. It allowed for dynamic additions and changes to the schedules throughout building the schedules through the algorithm. It was very efficient at memory utilization, as nothing needed to be pre-allocated for the schedules, resulting in a faster computation time. It also allowed for much faster access time, to increase the overall speed of the algorithm [Anon., 2015].

How the ScheduleGenerator class builds each schedule was the biggest challenge, as it needed to be very quick, in the sense of building the schedules within of starting, and efficient, in the sense of being able to access Schedule objects instantaneously, with the LinkedList. It needed to achieve all of these goals because the user does not have time to sit around while the application builds each schedule and then finally displays them to the user. It needed to be displaying them within seconds after clicking on the "Generate" button. Most of the computations and the brain of the application, the algorithm, is in the ScheduleGenerator's generate method. This method works in the following way:

Algorithm ScheduleGenerator

Input: list of Course's *courses*, list of Commitment's *commits*, list of String's *periods*.

Output: LinkedList of Schedule's *schedules.*

```
searchCount <- 0
subCount <- 0
while schedules.count != size, do
    posSchedg <- Schedule
    subCount <- courses.count
    shuffle courses
    for each c in courses, do
      shuffle c.sections
      for each s in c.sections, do
        if s.numDays == 0, then
          posSchedg <- posSchedg.add(s)
          if s.getSubSecs.count == 0, then continue
          subCount <- subCount - 1
          s.subSec <- shuffle s.getSubSecs
          ss <- s.getSubSecs[arc4random_uniform(s.getSubSecs.count)]
          switch periods.count
          case 2
            if commitConflicts(ss) && ss.getTimes[0].period == periods[0] ||
            ss.getTimes[0].period == periods[1], then
              if !posSchedg.add(ss), then
                posSchedg <-
                posSchedg.getSections.remove(posSchedg.getSections.count - 1)
            break
          case 1
            if !commitConflicts(ss) && ss.getTimes[0].period == periods[1], then
              if !posSchedg.add(ss), then
                posSchedg <-
                posSchedg.getSections.remove(posSchedg.getSections.count - 1)
        break
    default
        if !commitConflicts(ss), then
            if !posSchedg.add(ss), then
                posSchedg <-
                posSchedg.getSections.remove(posSchedg.getSections.count - 1)
        break
    if posSchedg.contains(s), then
        break

switch periods.count
case 2
    if !commitConflicts(s) && s.getTimes[0].period == periods[0] ||
    s.getTimes[0].period == periods[1], then
        if posSchedg.add(s), then
            if s.getSubSecs.count == 0, then continue
            subCount <- subCount - 1
            s.subSec <- shuffle s.getSubSecs
            ss <- s.getSubSecs[arc4random_uniform(s.getSubSecs.count)]
            if !commitConflicts(ss) && ss.getTimes[0].period == periods[0] ||
            ss.getTimes[0].period == periods[1], then
                if !posSchedg.add(ss), then
                    posSchedg.getSections.remove(posSchedg.getSections.count - 1)
    break
case 1
    if !commitConflicts(s) && s.getTimes[0].period == periods[0], then
        if posSchedg.add(s), then
            if s.getSubSecs.count == 0, then continue
            subCount <- subCount - 1
```

```
                    s.subSec <- shuffle s.getSubSecs
                    ss <- s.getSubSecs[arc4random_uniform(s.getSubSecs.count)]
                    if !commitConflicts(ss) && ss.getTimes[0].period == periods[0], then
                        if !posSchedg.add(ss), then
                            posSchedg <-
                            posSchedg.getSections.remove(posSchedg.getSections.count - 1)
                break
            default
                if !commitConflicts(s), then
                    if posSchedg.add(s), then
                        if s.getSubSecs.count == 0, then continue
                        subCount <- subCount - 1;
                        s.subSec <- shuffle s.getSubSecs
                        ss <- s.getSubSecs[arc4random_uniform(s.getSubSecs.count)]
                        if !commitConflicts(ss), then
                            if !posSchedg.add(ss), then
                                posSchedg <-
                                posSchedg.getSections.remove(posSchedg.getSections.count - 1)
                break
            if posSchedg.contains(s), then break

    if posSchedg.size == (courses.count*2 - subCount), then
        if schedules.count == 0, then
            schedules <- schedules.append(posSchedg)
        else if schedules.count > 0, then
            amountNotEqualTo <- 0
            for each index in schedules.count - 1, do
                if schedules[index] != posSchedg, then
                    amountNotEqualTo <- amountNotEqualTo + 1
            if $amountNotEqualTo >= schedules.count$, then
                $schedules$ <- $schedules.append(posSchedg)$

    if $searchCount >= size*10$, then break
    else then $searchCount$ <- $searchCount + 1$
    return $schedules$
```

From instantiating the Schedule object near the beginning of this paragraph, up to the beginning of this sentence represents one cycle of one possible schedule. If there was any conflicts found during the comparisons between Courses, Sections, SubSections, and Commitments, the cycle would be restarted with a new set of random Courses, Sections, SubSection objects. This cycle would then be repeated until either a maximum of 10 possible schedules were found, or until this cycle was repeated 100 times.

The reason for the random shuffling rather than just building a binary tree of all the possible Course, Section, and SubSection combinations is simply due to the fact that building a binary tree for this type of issue can take exponentially long, therefore avoiding this by shuffling the Courses, Sections, and SubSections randomly together, to the maximum bound of 100 times.

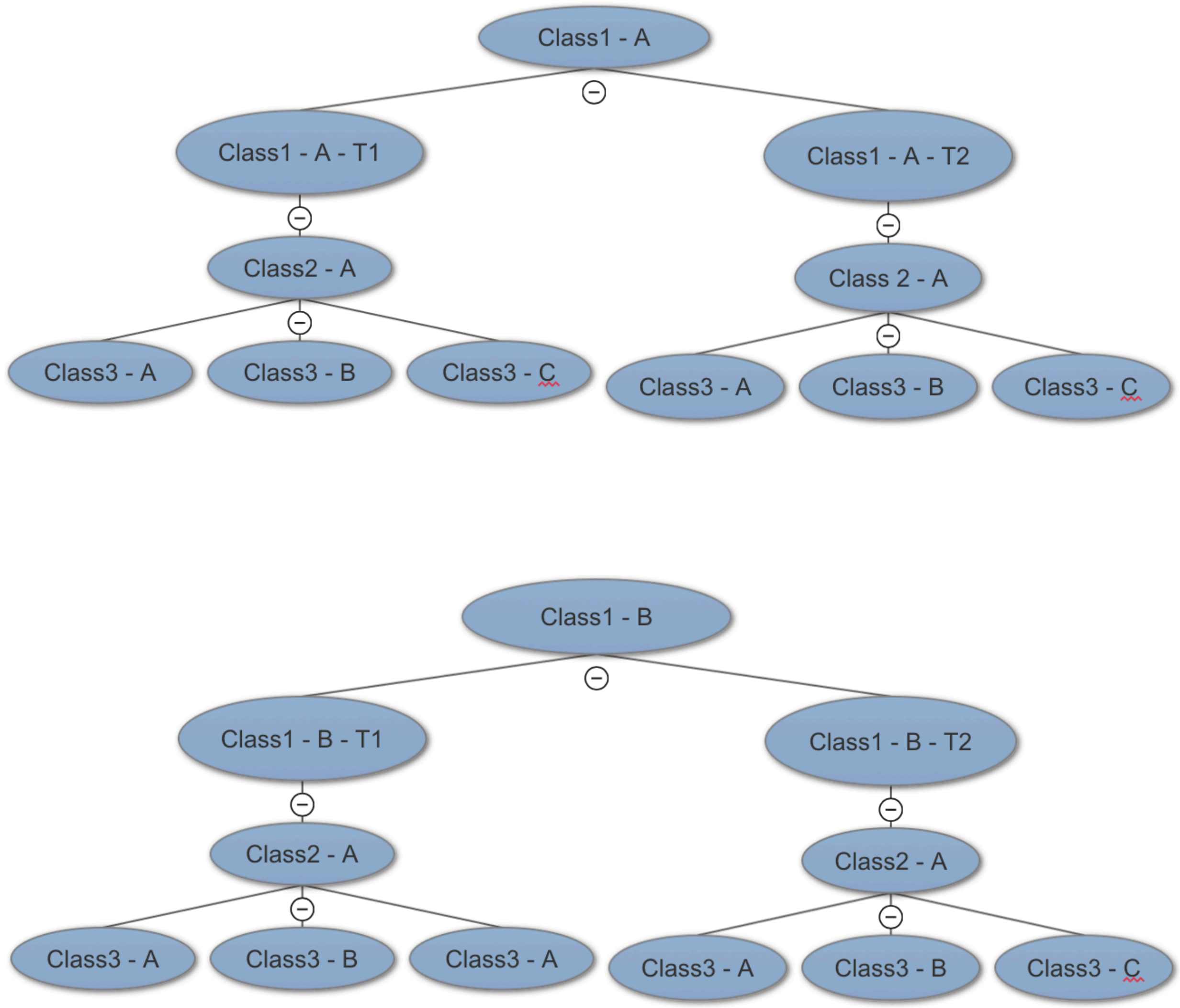


FIGURE 10 Binary tree for all of the possible combinations for the Course object, Class1, with Section objects, A and B, with SubSection objects, T1 and T2 for each Section object, Course object, Class2, with a Section object, A, and the Course object, Class3, with the Section objects, A, B and C.

When building a binary tree of all of the possible combinations, this is a very lengthly process that can take a lot of time to build each combinations. On top of the time it takes to build each and every Course, Section and SubSection combination, every single possible combination then needs to be traversed in some way to determine whether that combination has any conflicts or not. This can take exponential time, which was not an option due to the amount of time this would take to generate all of the possible schedules. Since scheduling is NP-Hard, to achieve all of the possible schedules being generated within seconds, shuffling the objects in the algorithm randomly was turned to [Safia, 2004]. Shuffling each of the objects together with a random integer allowed for the algorithm to be able to output the correct possible schedules 100% of the time during the limited testing (refer to Algorithm Testing section), with reducing the running time and providing an output of all the possible schedules, based on the limited testing that was done [Landman and Williams, 2016]. Through the tests of the algorithm, it generates all of the possible schedules within 100 cycles of the generate function, hence the reason for the 100 cycles being chosen as the cycle limit for the generate function explained at the end of the paragraph before this. This was tested by generating multiple different schedules, many different times with many different cycle limit numbers, resulting in 100 being a very optimal number for generating all of the possible schedules. With further testing, this number could get even smaller for the optimized output of schedules. In regards to the quickness and efficiency of the algorithm, on average, it outputs all of the possible schedules in 0.2 seconds. This was tested by calculating the time it took in nanoseconds, from the algorithm started to when the algorithm ended.

Algorithm Testing

Overall, the testing of the algorithm wasn't extensive due to the time constraints of this project and will need further testing against speed and validity for further confirmation than what has been confirmed presently for speed and validity of this algorithm. For hard-coded tests of the algorithm, testing was done within Xcode's provided Unit Testing files, in which the speed of the algorithm was tested and the validity of the algorithm was tested. The testing that was done within the Unit Tests wasn't fully extensive testing, as the majority of the testing was done with the application during runtime, within a live trial, in which the algorithm generation times were documented for each trial and during each trial, the validity was error checked for each schedule generated against Carleton's courses through the live trial. Within the Unit Tests, the average time it takes for the algorithm to generate all of the possible schedules is 0.08 seconds and the validity that was tested, which was very little, confirmed 100% validity of the algorithm. For the live trials of the application, the trials can be seen in the in FIGURE 13 in the Appendices section. The live trials yielded an average 0.2 second generation time for the algorithm with 100% validity as well. This validity was computed by comparing Carleton's supplied course times against each schedule course times that were generated, and if there was any conflicting times, or course information, this was documented.

End Result

All in all, this project was a success. Not only have all of the goals and objectives outlined in the "Goals and Objectives" section above been achieved, they have been achieved in thought of the perspective of the users' expectations being set rather high. Firstly, an iOS application was developed that allowed for users to select specific courses that they want to take, select entire days that they are not available to attend class, specific times during specific days that they are not able to attend class, and select a preferred time of the day that they would rather attend class. From this selected data, this iOS application then would generate all of the possible schedules accordingly and display this to the user in a simple, understandable, and readable way. Secondly, all of the possible schedules being generated were generated by a very complicated algorithm that was built from the ground up through lots of research and even more trial and error. The algorithm that was built worked quickly and efficiently while still outputting the most optimal solution. Therefore, when generating these schedules, the user did not have to wait around for the the algorithm to be finished, as the algorithm finishes, on average, in 0.2 seconds.

Overall, these two goals that were achieved, making this project, took well over 400 hours of testing, development, and research. With a considerable amount of testing throughout this projects lifespan from each individual user interface page of the application, to the inputs and outputs of every method, to the algorithm, I believe that this application is ready to help users build their course timetables more easily, quickly, and efficiently than ever.

Conclusion

Having achieved all of the initial goals stated, reaching these goals was a long process of careful consideration of alternatives and making a decision based off of those considerations when running into issues during the development of this project. For example, one of those issues that was ran into was attempting to generating all of the schedules for the user. This was initially attempted by building all of the possible combinations of schedules in a binary tree. Then from this binary tree, the algorithm would perform a depth first search on all of the possible combinations, whilst calculating contradictions, and finally outputting all of the possible schedules to the user. This was not an option as the algorithm was not able to output all of the possible schedules to the user within the stated goal timeframe of the user requesting them. The alternative to this problem was building an algorithm that used randomness to quicken the algorithm up to generating all of the possible schedules in 0.2 seconds, on average.

Limitations

The first limitation of this project was realized in the beginning design stages. That limitation being that this project is limited to the universities/colleges that agree to provide their course information for use by this application. This course information is not public information, therefore, the only way for this course information to be obtained from university/colleges is to request it from them. Alongside this, most universities/colleges do not keep the same courses, course times, course sections, course labs, course tutorials, and course instructors that are offered from semester to semester for long periods of time, therefore this information would need to be requested from universities/colleges at the beginning of every year for this application to keep up to date with the latest course information. At the time of writing this, the only university/college that agreed to provide their course information was Carleton University and they only agreed to provide the course information for the 2016/2017 Fall and Winter semesters. Therefore, this application at the time of writing this is limited to only being able to generate schedules with courses offered in the 2016/2017 Fall and Winter semesters at Carleton University. A potential solution to this limitation is reaching out to as many universities/colleges as possible and attempting to gain special access to their course information, or ask to be given the course information at the beginning of each school year.

Another limitation of this project was found in the early development stages of this application. That limitation being that all of the course information being parsed with the XMLParser is being stored locally on the device. Since all of the course information for the user chosen university/college is being parsed line-by-line, this can take quite some time. For the current course information that is available to the application, which was explained at the end of the previous paragraph, it can take up to 25 seconds to parse all of the course information and store it locally. A potential solution for this limitation would be to use a database instead, from which the application will only access the course information it needs, when the user selects that specific course. This would have been implemented but due to the timeframe of this project and the lack of my knowledge of implementing and using a database, there was not enough time for me for me to fully implement this, therefore I had to use what I had implemented already, the XMLParser.

Future Improvements

From the limitations stated above, a significant improvement of the application would be having access to course information for more the universities and colleges. This would result in the application being able to be used for as many universities and colleges that would be willing to provide their course information and would significantly improve the user base for this application as well. Also, another improvement for this application would be to implement a database that would hold all of the course information. Then to modify the application to only access this database for the course information that it needs at that particular time. This would eliminate the 25 second wait time for all of the course information for a particular university/college to be parsed and stored locally.

Another improvement of the application comes from the algorithm. Since the algorithm uses randomization, which was explained in depth in the ScheduleGenerator section, this opens up the possibility of the schedules that are being displayed to the user, being different if the possible schedules that are generated exceed 10. Meaning that if there is 12 schedules that are generated, and only 10 of those schedules are being shown, it is non-deterministic which of those 10/12 schedules will be displayed to the user due to the randomization factor of the algorithm. This can be resolved potentially by making the algorithm linear-random, in which the algorithm would follow some type of linear condition, whenever it would generate schedules. This would ensure consistency in the schedules outputted to the user for an overall improvement to the application.

Contributions to the Field

Since the goals of the project are particularly different from any type of field of study related to this project, there was not any significant contributions to any field related to this project. In regards to the application, it could potentially contribute to the motivation of more universities and colleges offering a better solution to help students build their course timetables by offering a tool related to the idea of this application. In regards to the algorithm, this algorithm was built solely off of the research of previous work that has been done already, therefore it is nothing innovative or new contributing to the field of algorithms. From the proposal of this project, which is still research-based, the main focus of it was to concentrate more on the concrete side, rather than the theoretical side. This project's sole purpose was geared towards developing an application that makes the process of building a course schedule for a university/college quicker, easier, and more efficient for the user. Overall, I believe this project could aid future student's in their decision making for certain user interface objects and structuring their iOS applications, and possibly extend on the work that I have done so far.

Concepts and Technologies Learned

Before building this application, I had not built any type of iOS application that was a fully fledged, working product. All of the iOS applications that I had developed up until starting to develop this project were very singular use and simple applications. For example, the iOS application that I had solely developed and I was most proud of up until starting this project, was a simple Tic-Tac-Toe game that allowed for two people to play Tic-Tac-Toe against each other on the same device. Building this application has taught me how to bring multiple smaller, separate partitions of an application idea together, and integrate each of those partitions within each other properly to build a fully fledged, working product. Also, before building this application, I had only taken one course on algorithms, in which the main focus in the course was studying and learning about previously built, famous algorithms, such as Dijkstra's algorithm. This course is what first introduced me to algorithms, therefore my knowledge of algorithms, the concepts of an algorithm, and building algorithms were very limited. Building the algorithm for this application forced me to learn about algorithms in a more in-depth perspective, to learn about building algorithms, to learn about specific aspects of an algorithm that will aid in your result (e.g. the use of randomization), and to learn about which data structures to use and when for optimized performance.

Appendices

File Contents

Provided with this project is a file containing the following:

1. Schedula
    - a. Schedula source code
    - a. Schedula file for the project configuration data and links to files referenced
    - b. Schedula logos used
2. PDF project report

Development Environment

The Apple provided software for developing applications for Apple products, Xcode, was used for the entire project.

Project Setup

In Xcode, click *Open another project* then browse to the directory containing the project. Select *Schedula* and click *Open*. Now Xcode should load all of the appropriate files and configurations automatically for you and you will now be ready to run the application.

Deployment and Testing

To run the application, click on the device, which is located to the right of the play button, and select a specific Apple device. The Apple devices that are supported by this application at the time of writing this is the iPhone 4S, iPhone SE, iPhone 7, and iPhone 7 Plus. Once selecting one of the previously listed devices, click on the play button. This play button will first start the associated emulator for the device that was selected, then it should automatically build and run the application on the emulator once it has fully loaded. If the application is not automatically built and run by the emulator, which seems to happen every once in a while due to some background emulator issues, clicking the stop button and play button again in Xcode should build and run the application properly. The Unit Testing for this application can be run by selecting the "Test Navigator" button located just below the play button and it looks like a yield sign. Once in the test navigator menu, select the play button directly beside the file named SchedulaTests. This will now run all the test functions and display the execution times and validity in the console.

Questionnaire

The questionnaire consisted of one question "Do you provide a tool to your students that helps with course scheduling?" This questionnaire was sent to 18 different universities in Ontario in an email. The following are the results of the questionnaire:

University Name	Yes	No	Unanswered
University of Ottawa	X		
Queen's University			X
McGill University		X	
Western University	X		
Carleton University		X	
Concordia University			X
University of Guelph		X	
University of Toronto			X
Ryerson University		X	
University of Waterloo			X
Brock University			X
Algoma University		X	
Lakehead University		X	
Trent University	X		
Wilfrid Laurier University	X		
University of Windsor			X
Nipissing University		X	

University Name	Yes	No	Unanswered
Laurentian University		X	

FIGURE 11 Questionnaire results.

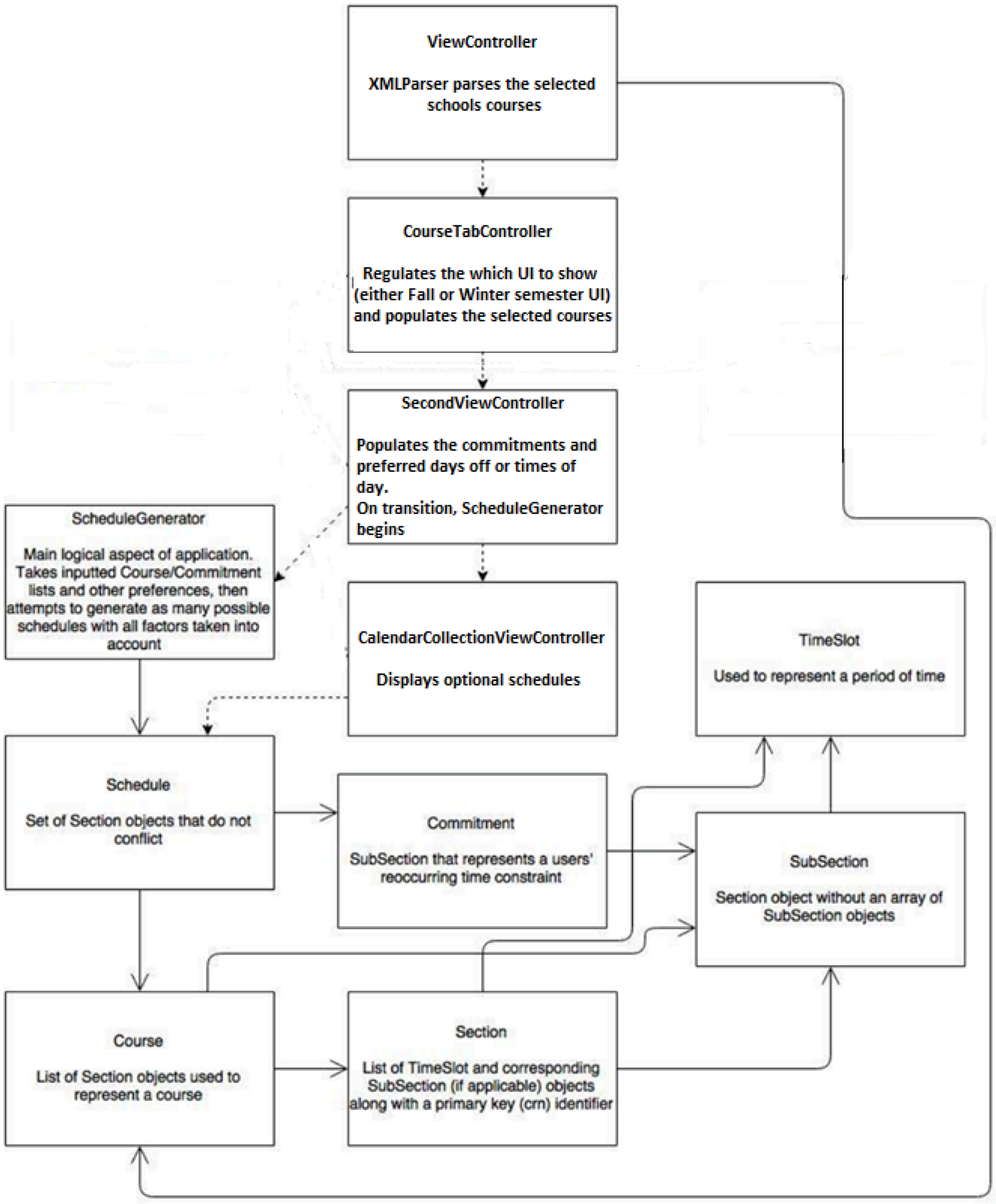

ViewController
XMLParser parses the selected schools courses
CourseTabController
Regulates the which UI to show (either Fall or Winter semester UI) and populates the selected courses
SecondViewController
Populates the commitments and preferred days off or times of day.
On transition, ScheduleGenerator begins
ScheduleGenerator
Main logical aspect of application. Takes inputted Course/Commitment lists and other preferences, then attempts to generate as many possible schedules with all factors taken into account
CalendarCollectionViewController
Displays optional schedules
TimeSlot
Used to represent a period of time
Schedule
Set of Section objects that do not conflict
Commitment
SubSection that represents a users' reoccurring time constraint
SubSection
Section object without an array of SubSection objects
Course
List of Section objects used to represent a course
Section
List of TimeSlot and corresponding SubSection (if applicable) objects along with a primary key (crn) identifier

FIGURE 12 Component model for the underlying structure between model and view controllers.

Course Set	Time (seconds)	Valid
COMP 2401, COMP 2402, COMP 2404, CGSC 1001, PHIL 2001	0.3	100%
PSYC 2001, COMP 3004, COMP 3007, COMP 3005, CGSC 2001	0.2	100%
COMP 1405, COMP 1805, PHIL 1200, PSYC 1001, CGSC 1001	0.4	100%
COMP 2401, COMP 2402	0.1	100%
COMP 3004, COMP 3005, COMP 3007	0.1	100%
COMP 4109, COMP 4004, COMP 4905, TSES 2305, ECON 1000	0.3	100%
SYSC 4106, ECON 1000, TSES 2305, COMP 1405	0.2	100%
SYSC 4106, ECON 1000, COMP 1405	0.2	100%
COMP 3004, COMP 3005	0.2	100%

FIGURE 13 Live testing chart through the application of the algorithm.

Bibliography

"Document Revision History." *The Swift Programming Language (Swift 4.0.3): Document Revision History*, 4 Dec. 2017, developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/RevisionHistory.html#//apple_ref/doc/uid/TP40014097-CH40-ID459.

"Linked List Advantages." *Learning Made Simple.* 19 Sept. 2014, www.c4learn.com/data-structure/linked-list-advantages/.

Premaratne, Pasan. "Should You Learn Swift or Objective-C? The Short Answer Is Both." *Treehouse Blog*, 13 Sept. 2016, blog.teamtreehouse.com/learn-swift-or-objective-c.

Randomized Algorithms. *Brilliant.org*. Retrieved October 14, 2017, December 5, 2017, https://brilliant.org/wiki/randomized-algorithms-overview/

Safia, Ahmed Abu. "Approximation Algorithms for Scheduling". 2004, www.math.mcgill.ca/vetta/CS760.dir/ahmed.pdf.

"Xcode Release Notes." *Guides and Sample Code*, 7 Dec. 2017, developer.apple.com/library/content/releasenotes/DeveloperTools/RN-Xcode/Chapters/Introduction.html.