Mert Yapucuoglu

Kelly Schombert

David Han

Brianna Vago

Kassandra Morando

**All In a Week’s Work**

**Programmer’s Documentation**

**Introduction**

This programmer’s documentation is designed to help programmers who aim to understand and modify the code after its initial completion. It defines and explains the main functionality provided by the program, and provides details on the parts that create the whole. The program can be best described using the six sections below.

1. Program Overview
2. Provided Functionality
3. Data Structures
4. File System Interactions
5. Code Files
6. Discovered Bugs
7. **Program Overview**

All in a Week’s Work program is developed specifically for the University of Oregon John E. Jaqua Academic Center, in order to replace the manual scheduling system with automated software. What makes this scheduling different than other software is that tutors and athletes have varying schedules that need to be matched in both availability and course to create a tutoring session. This is accepted into the system by the import of tutor and athlete information including identification, weekly availability, academic courses, and preferences.

The scheduling is done by accepting this information and running the scheduling algorithm at least 100 times to generate different schedules. In this document, a single run of a scheduling algorithm (which happens at least 100 times), will be often referred to as a “scheduling instance”. The difference between schedules created by “scheduling instances” is ensured by randomizing the following data for each different run of the scheduling algorithm:

* The order of the list that contains the athlete data instances
* The order that the courses are listed in tutor and athlete data instances
* The order of the available hours for each day in the tutor and athlete data instances

If what is being randomized is unclear, please refer to section 3. Data Structures, where the data classes are explained and the randomizing is elaborated further. The scheduling algorithm is also explained deeply in Scheduler.py in section 5.2, along with the scoring mentioned below.

With these randomizations, multiple different possible schedules are created, and the decisions made by the schedulers are scored by each step depending on the hour requirement of the athlete. By the end of each scheduling instance, the algorithm returns a score that will be used to determine the most optimized schedule, which then will be served to the user as the best schedule.

The program then saves the information of the best schedule to the program directory. Please refer to section 4. File System Interactions, for details on the file management.

With the existence of a built schedule, the UI then enables the user to provide a name and extract a separate file containing the schedule of that individual. This is done by reading the schedule saved in the program folder, filtering the sessions with the name, and then exporting the data to another file.

The individual schedule exporting functionality is provided if a schedule exists in the program folder, regardless of whether a new one is made. Meaning the user can terminate the program and open it back up again, while still being able to export individual schedules.

System specifications:

The system for this software is chosen as MacOS 10.14.6., and Python 3.7 is needed to run the code.

1. **Provided Functionality**

The program provides its functionality with a GUI that enables file import and export, and a scheduling algorithm that creates at least a hundred schedules and picks the best one.

* The GUI will present a button in the dropdown bar at the top left corner, which will allow for the user to import athlete and tutor information files to the system.
* Upon file import, the program will check if they are valid, and notify the user if otherwise.
* The ScheduleSystem will run the algorithms, and save the best schedule to the program files, notifying the user of completion.
* If a previous schedule done by the program exists or a new one is made, the GUI will show a name input area with which the user can enter a name to export the individual schedule for that tutor or athlete.
* The individual schedule file will be created in the program directory with the name indication who it belongs to.
* The user then will be notified upon the creation of the individual schedule file.
* The main schedule created will persist between program instances, enabling individual schedule export, unless a new one is made.

1. **Data Structures**

The system uses several data structures and data classes to manage the information it has. Structures described in the Structures section are separate data entities in an object class that is not solely for data management, whereas structures in the Classes section are components that are for keeping and organizing data that does not need special attention unlike the structures in the Structures section.

* 1. Structures

1. **required**

Is a priority queue imported from the “queue” library from standard python libraries. One instance of “required” exists inside every scheduler instance defined in Scheduler.py (see section 5.2). It is responsible for keeping athlete instances (see Classes section below) that have required tutoring hours. The ordering of this list is done by using the hour need as the priority, higher hour need resulting in more priority.

The priority of athletes with the same hour need (there is a lot of athletes that have the same hour need), is randomized by adding a unique decimal value to the hour need, thus randomizing their order between athletes of that same hour need. This randomization is done in each scheduler object instance, meaning the algorithms look at the order of students differently each run.

1. **optional**

Is another priority queue like “required” above, imported from the same library. It also exists singularly in each scheduler instance. and is responsible for keeping athlete instances (see Classes section below) that have optional tutoring hours.

The ordering is done the same way as “required”, and the hours of the athletes that share the same requirement are randomized each scheduling run the same way by adding a unique decimal to the hour.

1. **athleteList**

Is a list existing in each scheduler instance that holds the athlete object instances. It is used to create “required” and “optional”.

1. **tutorList**

Is a list existing in each scheduler instance that holds the tutor object instances, see the Classes section below for details on Tutor class.

1. **appointments**

Is a list existing in each scheduler instance that holds all the tutoring sessions scheduled by that instance. These session informations are stored as data classes called “appointment”, see the Classes section below for details.

* 1. Classes

1. **Athlete**

A separate data structure class that contains all the information about a single athlete. It accepts a dictionary of data on initialization and takes that data apart to store them separately as:

* self.id : int
* self.name : str
* self.lastname : str
* self.availability : list[list[int]]

Is a list of lists that contain integers, each list denoting the times the athlete is free for a tutoring session each day.

* self.subjects : list[str]
* self.hours : int

Amount of hours either required or desired by the athlete.

* self.required : bool

Is a boolean to decide whether the athlete will be placed in the required or optional queue (see Structures section above) in the scheduling instance. Its value is decided by the FileI/O module when the student GPA and year information is read. The GPA and year information is not stored to make it simpler, so the class accepts a boolean.

Then another variable is created to keep the hours left on each subject, this is done by a method (createHours()) explained in section 5. under the athlete section. It is a list that contains a tuple of subject and hour of tutoring left in that subject.

* self.hoursLeft : list[(subject,hour)]

Athlete instances are created individually in each scheduler instance and the self.subject list containing course names for that athlete is shuffled, meaning the order in which the items in it are contained are changed randomly, creating a different order of checking.

This is also the case for the lists contained in self.availability. Each list inside the list is shuffled to provide a random availability checking for each day.

1. **Tutor**

Another separate data structure class containing all the information about a single tutor. It accepts a dictionary of data on initialization and takes that data apart.

* self.id : int
* self.name : str
* self.lastname : str
* self.availability : list[list[int]]

Is a list of lists that contain integers, each list denoting the times the athlete is free for a tutoring session each day.

* self.subjects : list[str]
* self.hours : int

Max amount of hours that the tutor is willing to work for.

Tutor instances are created individually in each scheduler instance and the self.subject list containing course names for that tutor shuffled like it was done in the athlete class. This also applies to self.availability, the lists inside the list are also shuffled to provide varying checking for availability hours.

1. **Appointment**

The data structure class containing all the information about a scheduled tutoring appointment. It takes all the values it needs on init.

* self.classroom : str
* self.time : int
* self.day :int
* self.tutor : Tutor
* self.athletes: [Athlete]

It is a list of athletes because more than 1 athlete can be present at a tutoring session.

* self.subject : str

1. **File System Interactions**

In this section, the files that the system interacts with, either by importing or creating, are listed and explained.

* 1. File Imports

There are 2 types of files that the system requires to create a schedule.

1. **Athlete Information File (.csv)**

This file will be created and structured by the user to fit the program requirements specified in SRS, and it will contain all the information about each athlete line by line in the following format:

*First Name, Last Name, ID, GPA, Year, Hours Wanted, Subjects, Availability*

* First name: alphabetical
* Last name: alphabetical
* ID: a number
* GPA: a number between 0 and 4
* Year: a number between 0 and 4
* Hours wanted: a number, maximum 8
* Subjects: any string, mutliple subjects separated by spaces
* Availiability: Numbers, hours in same day are separated by spaces, days are separated by /

1. **Tutor Information File (.csv)**

This file will be created and structured by the user to fit the program requirements specified in SRS, and it will contain all the information about each tutor line by line in the following format:

*﻿First Name, Last Name, ID, Hours Wanted, Subjects, Availability*

* First name: alphabetical
* Last name: alphabetical
* ID: a number
* Hours wanted: a number, maximum 25
* Subjects: any string, mutliple subjects separated by spaces
* Availiability: Numbers, hours in same day are separated by spaces, days are separated by /

* 1. File Exports

There are 4 types of files that the system can export to the program directory,

1. **Main Schedule (schedule.csv)**

This is the main output of the program, containing the information on all the appointments in the best schedule, in a formatted way to make it look like a calendar. Each day/hour cell with an appointment will be in the following format:

*Athlete, Subject, Tutor, Classroom*

1. **Appointments (.txt)**

This file is for the system to re-use if the program is terminated and then ran again, and the user wants to export individual schedules. The system will use this file, which contains appointment information of all appointments of the most recent schedule, in order to filter the appointments of the desired person. It will be formatted in the following format:

*Hour, Day, Athlete, Subject, Tutor, Classroom*

1. **Individual Schedules (name\_lastname.csv)**

This file is generated by the user’s request, and it will contain the all appointment information of an individual athlete or tutor, in the same format as the Main Schedule.

1. **Errorlog (errorLog.txt)**

This file is a conditionally generated depending on errors in the input files. If more than 1 error is present in the tutor or athlete information input files (4.1.1 or 4.1.2), the the error and the line it occurs in the document is written to the errorLog.txt.

1. **Code Files**
   1. **ScheduleSystem.py**

This module is the highest level entity in the system and it connects other components together. It contains the ScheduleSystem class which initializes Manager Interface, FileIO, and Schedule instances, and calls their methods to manage the drive of the program.

* + 1. **\_\_init\_\_(self)**

This method creates the instance of ScheduleSystem. It creates a FileIO instance and uses its readSave() (see 5.4.3) function to try to read an existing Appointments type(see 4.2.2) file. If it exists, it is brought and saved to self.bestSchedule, and self.scheduleExists is set to True. It then creates a ManagerInterface(MI)(see 5.3) instance, passing it 2 functions of the ScheduleSystem class: signalSchedule() and extractIndividual() (see below for both and see section 5.3 for their usage). It also passes the status boolean, self.scheduleExists, so that MI knows the state of the program. It then waits for the MI to call one of the functions or termination.

* + 1. **\_signalSchedule(self, athletePath, tutorPath)**

This method can be called by the ManagerInterface(MI) instance upon new file imports by the user in order to create a new schedule. It uses the FileIO function readFiles() (see 5.4.1) and attempts to read the files. If the read is successful two lists of data is returned, if failed a False bool is returned which is returned to the MI to show read failure to the user.

Upon success, the \_createSchedules() (see below) function is called to create multiple scheduler instances and runs the algorithms. It then calls getBestSchedule() (see below) to use the score values of each scheduler and find the best one. It finally calls writeFiles() (see 5.4.2) to create save files, and returns True for the MI to notify the user of completion.

* + 1. **\_createSchedule(self)**

This method is called by signalSchedule() (see 5.1.2) in order to create schedulers and run their algorithms. At least 100 Scheduler instances are created and passed 3 parameters, self.athleteDataList, self.tutorDataList, and classrooms. Then each of the instances’ makeSchedule() (see 5.2.2) method is called to generate each schedule. Once the methods finish, the schedulers are added to self.schedules list of the ScheduleSystem object.

* + 1. **getBestSchedule(self)**

This methods loops through all the schedulers in self.schedules and finds the one with the highest score value. It sets it as the self.bestSchedule value of the ScheduleSystem class.

* + 1. **exportIndividual(self, name)**

This function will be called by the ManagerInterface function generateIndividual()(see 5.3.6) when the user decides to export an individual’s schedule. It will be passed a name parameter, and the function will go through each appointment in the self.bestSchedule’s appointments list, and find each appointment that contains that individual. It will then save those appointments to a list and call individualSchedule()(see 5.4.5) from fileIO to create a schedule .csv file for that individual.

* 1. **Scheduler.py**

Contains the object class Scheduler, which is responsible for keeping the information of a schedule, creating priority queues of tutors and athletes to help with the algorithm, and making the schedule with the information while scoring the decisions it makes.

* + 1. **\_\_init\_\_(self, athleteDataList, tutorDataList, classrooms)**

Creates a Scheduler object, saving each parameter to its respective self. value. Creates self.tutorList and self.athleteList using \_createLists()(see 5.2.7), which goes through the data list for athlete and tutor and creates Athlete and Tutor instances, adding them to self.athleteList and self.tutorList. Then \_createRequired and \_createOptional() are called to create priority queues self.required and self.optional. These queues are mostly unique to each scheduler instance, which is explained in 5.2.5-6.

* + 1. **makeSchedule(self)**

This function is called by the ScheduleSystem module function \_createSchedule (see 5.1.3) upon user file import. It calls two functions of the Scheduler class, \_scheduleRequired() and \_scheduleOptional() (see below for both) to build the schedule. It then returns the score of the scheduling back to the ScheduleSystem.

* + 1. **scheduleRequired(self)**

This function uses the self.required priority queue and goes through the queue creating 1-hour appointments for athletes in the order they are in the priority queue. It takes the athlete, goes through their availability each day, checks their subjects, finds a matching tutor with availability and subject, and creates an appointment and adds it to self.appointments list. If it can create an appointment, the hour need of both the tutor and the athlete is reduced by 1, the time for that subject is decreased in the athlete’s subject list, the score is incremented by 1000, and the athlete is added back to the required queue of they have at least an hour of tutoring need.

If a one-to-one appointment can’t be made with a tutor, the existing list of appointments (self.appointments) are checked to see a matching one can be found. If so, the score is incremented by 500, and the hours of the athlete are adjusted the same way as above.

* + 1. **scheduleOptional(self)**

This function works exactly the same as scheduleRequired, but it uses the self.optional priority queue which contains athletes with optional tutoring needs. It also increases the score by 1 if an appointment is made.

* + 1. **createRequired(self)**

Is called by \_\_init\_\_, to create a priority queue with the athletes that require a certain amount of tutoring hours. The method loops through athletes in the self.athleteList, checks if the athlete has a required status, if so, generates a unique three-figure decimal between 0 and 1, and adds it to the hour variable of the athlete. It then adds the athletes to the required priority queue using their hour value as the priority. This way even with athletes that have the same amount of hours needed, their place in the priority queue is unique to each scheduler instance. Causing slightly different schedules to be generated by the system.

* + 1. **createOptional(self)**

Is called by \_\_init\_\_ right after createRequired(), and it does exactly the same thing but it adds the athletes that do not have a required status to the optional priority queue.

* + 1. **createLists(self)**

Is called by the \_\_init\_\_ to loop through the tutor and athlete data and create Athlete and Tutor object instances using the data. The created tutors and athletes are added to self.tutorList and self.athleteList respectively.

* 1. **ManagerInterface.py**

This module contains the ManagerInterface class which is the GUI of the system. It creates and manages the content of tkinter windows to interact with the user. It accepts file inputs, name inputs for individual schedule creation, and shows status messages.

* + 1. **\_\_init\_\_(self,scheduleExists, signalSchedule, exportIndividual)**

Creates an instance of the ManagerInterface class and saves the boolean scheduleExists, and function pointers to signalSchedule() and exportIndividual(), which are passed functions of the ScheduleSystem module (see 5.1). It then calls the createDisplay()(see below) to initialize and fill the tkinter window that the user will use for interaction.

* + 1. **createDisplay(self)**

Creates a tkinter window, and sets its dimensions. Creates a Menu element and fills it with buttons that lead to documentation pages in the program files. Calls updateDisplay()(see below) to fill the main body with content.

* + 1. **updateDisplay(self)**

Called by createDisplay(), checks if the boolean passed from ScheduleSystem (self.scheduleExists) is True or False. If false, display the message to tell the user that there is no saved schedule, and display the button that upon press calls importAction() (see 5.3.5) to prompt file import to generate a new schedule.

If the boolean is True, it tells the user that a schedule is found, presenting the button for file prompt, along with a text input box and a generate button that will take a name input and call the function from ScheduleSystem called self.exportIndividual()(see 5.1.5) to create an individual schedule file.

* + 1. **startScheduling(self, filepath1, filePath2)**

This is called by importAction()(see below) function to pass import file paths to ScheduleSystem using its signalSchedule()(see 5.1.2) function and initiate generation and the making of schedules.

* + 1. **importAction(self)**

This is called when the user clicks the “generate new schedule” button on the ManagerInterface window. It prompts the user twice, first for the athlete data file and second for the tutor data file. It then saves the file paths and calls startScheduling()(5.3.4) while passing these paths to initiate schedule generation

flow.

* + 1. **generateIndividual(self)**

This method is bound to the “generate” button present at the UI when there is a made schedule. It will take in the input at the text input field and call exportIndividual()(see 5.1.5) to signal the ScheduleSystem for individual schedule file creation, which returns true or false indicating success or the name not found.

* 1. **FileIO.py**

Defines the FileIO class which contains the functions to provide file read and write service to the system.

* + 1. **readFiles(self, athFilePath, tutFilePath)**

Called by ScheduleSystem function signalSchedule()(see 5.1.2) after the user imports files into the system. The files are accepted as an absolute file path and are passed from the ManagerInterface to ScheduleSystem, and then to fileIO using this function. This function reads both athFilePath and tutFilePath, checks their fitness to the format required by the program, and returns False if they are not in the right format or if they don’t exist.

If the files are valid, it reads them line by line and saves the information in two lists of dictionaries, each dictionary containing information about athletes, or tutors. These lists are then returned back to the ScheduleSystem.

* + 1. **writeFiles(self, appointments)**

This function is called by the ScheduleSystem function signalSchedule()(see 5.1.2) after the scheduling is finished and the best schedule is chosen. It creates a calendar-like excel importable .csv file that contains all the appointments created for the schedule. It then calls writeSave()(see below) to create a separate save file for the schedule for easily schedule retrieval in future program runs.

* + 1. **readSave(self)**

ScheduleSystem object calls this function in order to check if a previously made schedule is saved in the program files. If it is, it is read and a list of appointments is created and returned back to the ScheduleSystem. If it doesn’t, a boolean of False is returned.

* + 1. **writeSave(self, appointments)**

Called by readFiles()(see 5.4.1) to create a separate Appointmentsu.txt file in the program files. This file will be used in order to easily retrieve the information about the most recently made schedule.

* + 1. **individualSchedule(self, appointments)**

Called by ScheduleSystem instance’s exportIndividual()(see 5.1.5) function to create a .csv file containing the appointments the person specified has. It will go through the appointments and create a calendar-like excel importable .csv format schedule for that individual in the program files.

* + 1. **createErrorReport(self, errorLog)**

Called by ScheduleSystem signalSchedule()(see 5.1.2) if the file import process returns multiple errors with the input, to create a errorLog.txt file to help the user fix their problems.

* 1. **Athlete.py**

Contains the Athlete data structure class which holds information about an athlete.

* + 1. **\_\_init\_\_(self, data)**

Creates an Athlete class instance by using the data in the data parameter. It uses random.shuffle(standard python library function from random) to shuffle the list of subjects in self.subjects.

It then uses shuffleTimes()(see below) to shuffle the hourly availabilities in each day of the week to provide unique randomness. This is done because the algorithm checks each day’s availability in order, and if the orders are different in each scheduling algorithm, different schedules will be produced, enabling the system to choose the best one.

Then createHours()(see below) is called to create a list of (subject hour) tuples that show how many hours of tutoring can be appointed to each subject. This is saved in self.hoursLeft, and is used by the scheduling algorithm.

* + 1. **shuffleTimes(self, availiability)**

Is called by \_\_init\_\_ to shuffle the availability hours of each day in the self.availability list of lists.

* + 1. **createHours()**

Is called by \_\_init\_\_ to go through each subject in self.subject list and allocate 2 hours of tutoring hour to each. Then, if any amount of tutoring hours is left, the rest is allocated to each subject 1 hour at a time.

* 1. **Tutor.py**

Defines the Tutor data structure class which holds information about a tutor.

* + 1. **\_\_init\_\_(self, data)**

Creates a Tutor class instance by using the data in the data parameter. It uses random.shuffle(standard python library function from random) to shuffle the list of subjects in self.subjects.

It then uses shuffleTimes()(see below) to shuffle the hourly availabilities in each day of the week to provide unique randomness. This is done because the algorithm checks each day’s availability in order, and if the orders are different in each scheduling algorithm, different schedules will be produced, enabling the system to choose the best one.

* + 1. **shuffleTimes(self, availiability)**

Is called by \_\_init\_\_ to shuffle the availability hours of each day in the self.availability list of lists.

* 1. **Appointment.py**

Contains the Appointment data structure class which contains the information for an hourly tutoring appointment between an athlete and tutor.

* + 1. **\_\_init\_\_(self, time, tutor, athlete, subject, classroom)**

Creates an instance of the Appointment class, and saves each parameter to its own self. value.

1. **Discovered Bugs**
   1. **The top number of hours for required athletes is 8, cant go up**