**SEAT Code Documentation (For SEAT v0.2.1)**

This document explains the code structure and purpose of every part of the software in detail. Please read this document thoroughly before making changes to the SEAT software.

All changes to be preferably made in the original Google Document, and not the local Microsoft Word version uploaded on Github.

Note that this documentation and the SEAT software, in either binary or source code form, is not to be distributed or used without explicit permission from the SEAT team, IIT Madras.

**SEAT Software Team:**

1. Akshay Utture - [akshay.utture@gmail.com](mailto:akshay.utture@gmail.com)
2. Meghana Nasre - [meghana.nasre@gmail.com](mailto:meghana.nasre@gmail.com)
3. Saranya Sundaresan Malarvizhi - [saranyahighflier@gmail.com](mailto:saranyahighflier@gmail.com)
4. John Augustine - [j.e.augustine@gmail.com](mailto:j.e.augustine@gmail.com)
5. Vedant Somani - [iamvedantsomani@gmail.com](mailto:iamvedantsomani@gmail.com)

Document Version:

This version of the document was prepared in July 2017, for the SEAT v0.2.1, which was used to make the July-Nov 2017 SEAT allotment (First Round).

**INDEX**

**A] RULES AND GUIDELINES TO USE THE SOFTWARE**

A1) Important Rules to Follow

A2) Important Guidelines to Follow

A3) Coding Conventions

**B] USER GUIDE (HOW TO USE THE SOFTWARE)**

B1) What does this software do?

B2) Important Terminology Used

B3) Input Files used by the Software

B4) Output Files given out by the Software

B5) Sample Input and Output Files

B6) Running the software from the terminal

B7) Running the software from the GUI

**C] THEORY**

C1) Theoretical Context of the Problem

C2) Algorithms used

C3) Evaluation Metrics

**D] CODE STRUCTURE**

D1) Pre-requisites to learn before trying to understand the Code

D2) Overview of Code Structure

D3) ‘models’ package

D4) ‘Main’ package

D5) ‘Services’ package

D6) ‘Tests’ package

D7) ‘SEAT-documents’ package

D8) Building the Project

**E] Appendix (Empty as of now)**

**A1) Important Rules to Follow**

The following set of Rules must be followed strictly.

1. The Source Code for SEAT as well as this Code Documentation must be kept confidential and not shared with members outside the SEAT team. This also includes the algorithms and allotment policy to be used.
2. The software must be saved online in a private repository in order to maintain confidentiality, and not stored on 1 person’s hard disk, where there could be a risk of losing the source code in the event of a crash. Source code backups should also be made.
3. While dealing with any kind of data used for the allotment, including but not limited to Student GPA Data, Student Registration Data, Course Data, it must be kept confidential and not shared with members outside the SEAT team.
4. Any changes to the allotment policy must be informed to the faculty in charge for the SEAT software (currently Professor Meghana Nasre and Professor John Augustine)
5. All modifications to the input or output files for the final allotment should be informed to the faculty in charge for the SEAT software.
6. Any piece of code added should be well commented, and documented in this code documentation.
7. If any of the above rules cannot be followed for whatever reason, permission must be taken from the faculty in charge for the SEAT software (currently Professor Meghana Nasre and Professor John Augustine)

**A2) Important Guidelines to Follow**

The following set of Guidelines should be followed while making changes to the Code.

1. Please do not attempt to use any programming language other than Java in the project. If you do not know Java, learn it. Mixing multiple programming languages gets very messy (will not be getting into the details of why), and requires any person modifying the software to know all the languages used.
2. As far as possible, functions should be limited to 100 lines of code. Anything larger should be split into functions.
3. Elaborate commenting should be done for any code added or modified.
4. Corresponding to the code added/modified, the corresponding sections in the code documentation should be modified.
5. Names should be as descriptive as possible. For example, if a function takes in an argument of a list and an element, and moves the element to the start of the list, the name of the function should be ‘moveElementToStartOfTheList()’
6. A version control system like GIT should be used so that multiple people in the team can make changes to the software, and for version control purposes.
7. The versions actually used for allotments should be marked using the ‘git version’ command.
8. Global Variables should be avoided as much as possible.
9. Unit test cases must be written for every new function added/modified, wherever possible.
10. The JUnit test cases must be run after modifying the software
11. Fields that are read from the input and that should not be modified should be made ‘final’, or ‘private’ with a getter() method and not setter() method.
12. Avoid unnecessarily creating interfaces or using inheritance. This is not a software library. It is a small software - keep it simple.

THE ESSENCE IS THAT YOU NEED TO REMEMBER THAT MULTIPLE PEOPLE WILL MODIFY THIS SOFTWARE SIMULTANEOUSLY. ALSO SOMEONE MIGHT MODIFY YOUR CODE 4 YEARS FROM NOW. WHATEVER YOU DO, KEEP IN MIND HOW EASY IT WILL BE FOR SOMEONE ELSE TO UNDERSTAND AND MODIFY YOUR CODE YEARS FROM NOW.

**A3] Coding Conventions**

1. Standard Java Coding Conventions to be Followed.
2. Code must be indented properly (Indentation using Tab (4 space tab) )
3. All names in Camel Case (google this in case you do not know). Hence the underscore symbol should be avoided in all variable names.
4. Names of Variables, Functions, and packages to start with a lowercase letter
5. Names of Classes to start with an uppercase letter.
6. Only one line of code on one line
7. Code must be elaborately commented. Give a high level understanding of what is being done, instead of giving low level details.
8. The body of a ‘if’ or ‘else’ or ‘for’ must be followed by a block in braces

Example. The following should be avoided

*if (condition) statement;*

Instead, we should use the following convention.

*if (condition){*

*Statement;*

*}*

1. The concept of high cohesion and low coupling should be followed.

High cohesion for a module or function means that all parts of the module or function do a single task. If a single function prints both per student statistics and the per student allotted courses, the function has low cohesion, because the various parts of the function are not doing a single task. If we keep 2 separate functions, 1 for printing per student statistics and 1 for printing per student allotted courses, we maintain high cohesion.

The coupling between 2 modules is high if changes in 1 module cause a large number of changes in another module. For example, if a change in the output format causes several changes in the algorithm code, then the 2 modules have high coupling. We would prefer to have low coupling, so that any change can be made only in a single place. Ideally, for low coupling, changes in the output format should not affect changes in any other module

**B] User Guide**

This section is available in the ‘SEAT User Guide’ document. It has not been repeated here to avoid duplication.

Read the ‘SEAT User Guide’ before moving on to the next section.

**C1] Theoretical Context of the Problem**

**Reading Material**

1. Stable Matching Problem (SM)
2. Hospital Resident’s Problem (HR)
3. (optional) Pareto optimality in many-to-many matching problems (K. Cechlárová, P. Eirinakis, T. Fleiner, D. Magos, I. Mourtos and E. Potpinková)
4. (optional) Classified Stable Matching (Chien-Chung Huang)

**Abstract Theoretical Problem being Dealt with**

We start of with a basic HR problem, and at each new bullet point, we keep modifying the setting until we have an abstract problem that matches the actual allotment situation in IIT Madras.

1. As in the HR (Hospital Resident’s problem) setting, we have students, and courses. Courses have capacities, and students wish to get allotted to courses. Both students and courses have strict preference lists ranking each other. This has an exact and polynomial time solution to find a stable matching.
2. Students have capacities. I.e. they can take multiple courses in one semester.
3. Courses have time slots, and for now let us assume that each course has just 1 slot, and that no 2 slots overlap. Now, a student can take at most 1 course in each slot. The problem is now of the Laminar Classified Stable Matching as mentioned in the Classified Stable Matching paper by Chien-Chung Huang (also given in the reading material). It still has an exact polynomial time solution to find a stable matching.
4. But in the actual setting, a course can have multiple slots, and 2 different slots can overlap in arbitrary ways. For example it is possible that, Slot A and Slot B clash, Slot B and Slot C clash, but Slot A and Slot C do not clash. This makes the problem a ‘Non-Laminar Classified Stable Matching Problem’, and as proved in the same Classified Stable Matching paper, it is an NP complete problem, and hence searching for a stable matching is futile.
5. Students can colour code all their preferences. If they give the same colour code to a set of courses, that student should get allotted at most one course from that set of courses. For example, if a student wants only one Maths course, he will colour all of his Maths courses with the same colour. The problem is still a ‘Non-Laminar Classified Stable Matching Problem’.
6. Adding another dimension of complication is that students have a credit limit every semester (unique credit capacities for every student). Each course has some credits, and if a student is allotted a set of courses, the sum of the credits of those courses should not exceed his credit limit. Hence, a course does not occupy a capacity of ‘1’ for a student, but a capacity equal to its credits, which could be an arbitrary number.

In this situation with credits, the definition of a stable matching itself is unclear, because it is possible to swap 1 course for multiple courses, and occupy the same number of credits of the student, or vice versa. Hence we do not attempt to use the notion of Stability here. Even if we did not have credits, the problem is still NP complete. Hence we have to devise our own set of algorithms to do the allotment, as well as our own ‘Evaluation Metrics’ to decide how good an allotment is, so that we can compare the allotments done by different algorithms.

**C2] Algorithms Used**

1. **Iterative Algorithm Framework**

This is a common framework used by all iterative type of algorithms. Currently this includes - Iterative HR, First Preference Allotment and Slot Based HR.

We will give a common algorithm and proof of constraints being met for the framework, and hence it will be applicable to all the algorithms under this framework. Only the ‘solve()’ method on Line 5 has to be specified for each algorithm.

Assume that no student gives a preference for a course whose credits exceeds his/her maximum credit limit.

**Algorithm**

*1. studentList = list of students*

*2. courseList = list of courses (each course has residual capacity = original capacity)*

*3. While (studentList is not empty)*

*4. {*

*5. solve() using the residual capcacities of courses;*

*6. For (student s1 in studentList)*

*7. {*

*8. C1 = course allotted to student s1 in line 5*

*9. Freeze the allotment (s1-c1)*

*10. Remove c1 and every fully occupied course from the preference list of s1*

*11. Remove every course from the preference list of s1 that conflicts with c1*

*12. Remove every course from the preference list of s1 that does not fit in the remaining*

*credits of the student*

*13.. If s1 now has an empty preference list, remove him from the studentList.*

*14.. }*

*15. Calculate the residual capacities for each course*

*16.}*

Note that in line 11, we mention that we remove courses which conflict with c1. This refers to all the courses which either

* Colour conflict with c1
* Slot conflict with c1 (they do not necessarily have the same slots. They can have slots that overlap for some time on some day of the week)

So the algorithm iteratively carries out the solve() function by considering each student to have a capacity of 1 at the start of the iteration. After each iteration, we remove all those preferences which have a conflict with the course allotted in the current slot, and fit in the remaining credits of the student.

The solve() function can be any one-to-many allotment function, and currently there are 3 algorithms implemented based on different functions for the solve() function. These are IterativeHR, FirstPreferenceAllotment and SlotBasedHR.

**Proof of Meeting Course Capacity Constraints**

The course capacities of a course are never violated, assuming the ‘solve()’ step never violates the residual capacities it is provided. Hence, the residual capacity for a course starts with the original capacity and never goes below zero. So no course can be allotted more students than it’s original capacity.

**Proof of Meeting Slot Constraints and Colour Constraints**

We prove that the slot conflicts and colour conflicts are satisfied using the following Induction Proof.

Inductive Hypothesis : A course allotted in the current iteration cannot conflict with a course allotted in any previous iteration

Base Case : During the first iteration of the solveHR(), each student gets allotted at most 1 course, and hence there is no possibility of conflicts.

Induction Case: In iteration ‘i’ a course ‘c1’ is allotted to a student ‘s1’. It cannot conflict with a course ‘c2’ allotted to ‘s1’ in round ‘j’ ( j<i ), because in round ‘j’ the Line 11 from the algorithm would have removed ‘c1’ from the preference list of the student ‘s1’.

**Proof of Meeting Student Credits Constraints**

We prove that the credit limits are satisfied using the following Induction Proof.

Induction Hypothesis : A course allotted to a student ‘s1’ in the iteration ‘ i+1 ’ cannot violate the credit limits of the students given that the student’s credit limit in iteration ‘ i ‘ was not violated.

Base Case : During the first iteration of the solveHR(), each student gets allotted at most 1 course, and using the assumption that no student gives a preference for a course whose credits exceeds his/her maximum credit limit, we can guarentee that the credit limit will not be violated.

Induction Case: In iteration ‘i’, there is no credit limit violation according to the Induction hypothesis. And in Line 12, we remove all courses from the preference list of ‘s1’ that do not fit in the remaining credits for ‘s1’. Hence in iteration ‘ i + 1 ‘ we cannot allot a course that will cause the credit limit to be violated.

**Time Complexity**

The outer while loop on line 3 can run at most O(m) times, where ‘m’ is the total number of preferences given by all students. Let ‘n’ = max(number of students, number of courses)

The ‘for’ loop from line 6-14 has a complexity of O(m), because we will iterate over each student preference once.

Overall Time Complexity =

Complexity of outer while loop \* (Complexity of for loop + Complexity of solve)

Therefore Overall Time Complexity = O(m) \* ( O(m) + Complexity of solve)

We might be able to prove a tighter time complexity bound in some cases, because we might be able to bound the number of iterations of the outer while loop based on the operations in the solve function.

**Proof of Pareto Optimality from the students side**

The paper “Pareto optimality in many-to-many matching problems (K. Cechlárová, P. Eirinakis, T. Fleiner, D. Magos, I. Mourtos and E. Potpinková)” mentions the conditions for a Pareto optimal many-to-many matching. Please read the paper for background for the rest of the proof.

Theorom 1-4 use the assumption that the solve() function is Pareto optimal among the subset of students who got allotted a course.

Theorem 1: Whenever a course is removed from a preference by the algorithm, let us call the remaining preference list as the reduced preference list. At every point of time in the algorithm, the set of courses preferred above those in the reduced preference list are either allotted, removed because they are not feasible due to some slot-clash or colour clash, insufficient credits or course capacity is full.

Proof: If not, then the course should have got allotted to the student in the iteration when it was at the top of the reduced preference list, because the solve() function is Pareto optimal to the set of

Theorem 2: Allotment given by the Iterative Algorithm Framework is Maximal

Proof: Since we end with an empty reduced preference list for every student in the end, there are no more additional student-course allotments that can happen. Hence the allotment is maximal. Theorem 1 establishes that the courses in outside the reduced preference list are not feasible.

Theorem 3: Allotment given by the Iterative Algorithm Framework is Trade-in Free.

Proof: Assume that the allotment is not Trade-in free. Let s1 be a student who wants to trade in the set of courses ‘C’ for the course c1.

Obviously c1 must be preferred over every course in C. Let c2 be the top preferred course in c1. From the fact that c1>c2 and c1 was not allotted and using the Theorom 2, we can conclude that in the round where c2 was allotted, c1 must have been removed because not being feasible due to some slot-clash or colour clash, insufficient credits or course capacity is full.

This contradicts the assumption that c1 could be traded in for ‘C’ because M(s1)\C U {c1} is not feasible. Qed by contradiction.

Theorem 4: Allotment given by the Iterative Algorithm Framework is Coalition Free.

Proof: If there is a coalition K = ((A0 , C0 ), . . . , (An,Cn )) in M, let Ai be the applicant at the earliest iteration in which the pairs in K were matched.

Case1: A(i+1)-C(i+1) was not matched in the same Iteration

This implies that Ai selected Ci instead of the more preferre C(i+1) , which, was selected by A(i+1) at a subsequent round ; the only reason for that to occur would be that C(i+1) had no empty slot, which in turns contradicts that C(i+1) was available for A(i+1) at a subsequent iteration.

Case2: A(i+1)-C(i+1) was matched in the same Iteration

(All arithmetic modulo n)

The entire coalition could not have been allotted in 1 iteration because we know that ‘solve()’ is Pareto optimal. Hence let (Ai-Ci, A(i+1)-C(i+1) ….. , Ak-Ck) be allotted, in which Ak-Ck was the last pair of the coalition in the sequence starting from ‘i’ to get allotted

Again following the logic of the previous case, the only reason Ak-C(k+1) was not matched, was because C(k+1) was full, but this contradicts the fact that A(k+1) got allotted C(k+1) in a later round.

Qed.

Theorem 5: The Iterative Allotment Framework gives a Pareto Optimal Matching from the Student’s side if the ‘solve()’ method is Pareto Optimal from the student’s side.

Proof: By Theorems 2,3,4 we know that the allotment is Maximal, Trade-In Free and Coalition Free, and using the Theorem from the paper “Pareto optimality in many-to-many matching problems”, we know that this is a sufficient condition for Pareto Optimality.

**2) Solve() Method for Iterative HR**

This method is exactly the Hospital Resident’s Gale Shapely based algorithm (Proposals algorithm), assuming that each student has a capacity of only 1, and capacities for the courses are the residual capacities

Time Complexity of HR = O(m)

Therefore, Time Complexity of IterativeHR = O

Can prove a tighter bound in this case, because in each iteration of solve, a student gets allotted to at least 1 course on his preference list, or exhausts his entire preference list.

Time Complexity of HR = O(m)

Number of iterations of outer while loop = O(L )

Time Complexity of IterativeHR = O(m\*L)

Where L = max over all students (number of courses that can be taken by a student).

Note that L will surely be lesser than the size of the largest preference list of a student.

**3) Solve() Method for First Preference Allotment**

In this method, we assume that each student has capacity 1 and capacities of the courses are the residual capacities. We temporarily allot each student to his top preference in the remaining part of his preference list. Then, for every course oversubscribed by ‘k’ students, we remove the worst ‘k’ students, as seen in the course’s preference list.

Time Complexity of one iteration = O(n)

Therefore, Time Complexity of First Preference Allotment = O(mn)

**4) Solve() Method for Slot based HR**

This method is exactly the Hospital Resident’s Gale Shapely based algorithm (Proposals algorithm), assuming that each student has a capacity of only 1, and capacities for the courses are the residual capacities, but with the following modification. We start of by picking an optimal ordering for the slots. In each call to the solve() method, we pick the next slot, and carry out the HR algorithm only for the preferences in that particular slot.

Time Complexity of HR = O(m)

Therefore, Time Complexity of IterativeHR = O

Can prove a tighter bound in this case, because the number of iterations of the while loop is lesser than or equal to the number of slots.

Time Complexity of HR = O(m)

Number of iterations of outer while loop = O(S )

Time Complexity of IterativeHR = O(m\*S)

Where S = number of slots

Getting an Optimal Ordering - Heuristic 1

Order the slots based on the number of first preferences for courses which run in that slot. (the course could additionally run in other slots as well)

Getting an Optimal Ordering - Heuristic 2

Maintain a score for each slot.

*for every student preference {*

*n = size of studentPreference list;*

*i = rank of student preference*

*add a score of (n-i+1) for every slot of the student preference*

*}*

The slots are now sorted based on this score. The central idea is that the slots which occour in courses which are higher on students’ preference lists should be given a higher score.

**C3) Evaluation Metrics**

Read the ‘Effective Average Rank’ and ‘Credit Satisfaction Ratio’ metrics in the SEAT User Guide.

**D] CODE STRUCTURE**

**D1) Prerequisites to learn before trying to understand the Code**

To be learnt before trying to even read the code

* Java (including Java Classes, Generics, Access Modifiers (public,private,protected), File Reading/Writing)
* Thorough understanding of the SEAT User Guide
* Finish reading the previous sections of this documentation

To be learnt before making any modifications to the code

* Git (for version control)
* Java (including Generics, Collections, Access Modifiers)
* JUnit Tests (for modifying tests)
* Java <check Graphics library> (for modifying the GUI)

**D2) Overview of Code Structure**

The flow of input data, as highlighted in the diagram below, shows how the input data is finally used to generate the output data.

* The various input files are read into Java objects
* An input sanitization is done to ensure that the input data has no errors. It tries to fix any errors if possible
* Run the allotment
* An output sanitization is done to ensure that the output data has no errors. All the constraints like slot clashes, course capacities, student credits, colour clashes,etc are checked.
* The output is printed in various formats. Also several statistics for the allotment are also printed



Here is a high level explanation of the various packages and files in the root directory

* Main : This package contains the files which host the main or the driver function for the whole software. In the main function, we simply call the various high level functions sequentially (one after another) - readInput(); sanitizeInput(); runAllotment(); sanitizeOutput(); printStatistics(); printAllotment();
* Models : This package contains the various entities, with 1 class for each type of entity. Eg. student, course, slot
* Services : This package contains the implementations of the various functions that are called from the main function, like readInput(), printOutput(), runAllotment(), etc
* Tests : This package contains the test cases, with their expected output, as well as some JUnit tests for unit testing
* SEAT-documents : This folder is a collection of important documents for SEAT, like this Code documentation
* README.txt : A simple README
* Manifest.mf : A config file that specifies which Class file has the main file to be executed, when we are generating the executable JAR for this software
* Makefile : Used for compiling (building) the project into an executable JAR

**D3) ‘models’ package**

This section explains the various files in the ‘models’ package. There are 9 Classes here.

Notice that it most of the files, some fields are declared private, because they are not to be modified after the student object is created (happens during the reading of the input files). This is because we do not want these variables to be modified during the course of the execution (especially during the main allotment algorithm), because we will be using these fields while computing some of the statistics or printing some of the outputs. If you must modify a private field, make a copy of it and modify the copy (remember to make a deep copy for arrays). The private fields can be read using the getxxxx() methods

Fields, functions or files that have not been explained here should be obvious to understand from the comments in the file.

**a) Student.java**

This class represents a student. Please read the Student.java file and all the comments in the file. Now read some of the important points regarding this file

* Notice that there are 3 variables related to the credits of a student in the sem.
* public int maxCreditsInSem; //The maximum credits that the student has stated he can take. It includes both elective and core credits. Read only.
* private int maxElectiveCreditsInSem; //The maximum elective credits that the student can take. Will be calculated by subtracting the core course credits from 'maxCreditsInSem'. Read only.
* public int electiveCreditsLeft; //The elective credits left at any point of time. This can change during the algorithm as this student gets allotted courses.
* Each student has 4 lists for the StudentPreference objects. The purpose of these 4 is as follows :
* studentPreferenceList: //Original preference list. Will not be modified once after the Input Sanitization step.
* coursesUnallotted : //Set of courses from the preference list which the student can still apply to at any point of time. Can be modified during the algorithm. Is initally set to the studentPreferenceList
* orderedListOfcoursesAllotted; //List of courses from the preference list which the student has been allotted to. Can be modified during the algorithm. It is an ordered list in the sense that the order of the arraylist should maintain the order in which the allotments to these preferences was made. This order is important for the 'ReasonsForNotAllottingPreference' module, where we iterate through the ordered list of allotted courses to compute why a particular elective was not allotted.
* public ArrayList<StudentPreference> invalidPreferences : //Set of courses from the preference list which are invalid (could be because of a slot clash with a core course, or that the course takes more credits that the student's maxElectiveCreditsInSem. Not modifiable after Input Sanitization.
* Hence, any course on the student preference list input file either goes into studentPreferenceList if it is valid, or into invalidPreferences if it is invalid. These 2 lists are not modifiable and are maintained for the purpose of statistics. coursesUnallotted is a copy of studentPreferenceList which can be modified as needed in the course of the algorithm. Courses that get allotted during the course of the algorithm get added into orderedListOfcoursesAllotted.
* effectiveAverageRank and creditSatisfactionRatio are fields which are used for statistics. They will be explained in the services.GetStatistics module section.

**b) Course.java**

* There are 2 fields ‘courseNumberToPrint’ and ‘courseNumber’ for the following reason. We will split a course ‘CS1000’ into 2 parts - ‘CS1000$outside’ and ‘CS1000$inside’, and hence the 2 parts will have a separate course number. But, while printing the final output, we do not want the ‘$inside’ to be printed, and hence we store the original name of ‘CS1100’ in the field ‘courseNumberToPrint’
* There are 2 fields for capacity

- capacity; //The number of students the course can accomodate as given in the input file. Read only.

- capacityStillFree; //The capacity left in the course. Will be modified during the course of the algorithm as the students get allocated to it. Can be modified. Initially set to capacity

* There ‘currentIterationStudentAllottedList’ is a temporary variable to be used during the algorithm to store a reference to a student who is allotted in the current iteration of the algorithm. Has no significance before and after the algorithm module executes.

**c) Slot.java**

Represents a slot in the timetable like ‘A slot’, ‘B slot’, etc. It has a name and a list of class timings through the week. Checking if 2 slots clash is done by taking every 2 pairs of slot timings of the slots, and checking if the time period between the start time and end time of the class timings overlaps.

**D4) ‘Main’ package**

This package contains the ‘main’ function or entry point for the software and orchestrates (manages) the calling of various modules in the correct order. There are 2 versions of the software - 1 with GUI and 1 without GUI, with an entry point for each of them - MainWithGUI.java and MainWithoutGUI.java. As of writing this documentation, the GUI is not ready yet. Each of the 2 entry point files simply take in all the input file or folder names, and call the function ‘ExecuteStepsForAllotment.executeAllotmentSteps()’

The ‘ExecuteStepsForAllotment.executeAllotmentSteps()’ orchestrates the calling of the modules in the correct order. The order in which the modules are called is as follows

1. CheckInputFormats
2. DataInput
3. GenerateCoursePreferences
4. It calculates elective credits for each student - done by executeAllotmentSteps() function itself
5. InputSanitization
6. Runs the main algorithm (either IterativeHRalgorithm, FirstPreferenceAllotmentAlgorithm or SlotBasedHRalgorithm).
7. OutputSanitization
8. GetStatistics
9. ReasonsForNotAllottingPreferences
10. ExchangeUnstablePairs
11. DataOutput

From the past few allotments we have concluded that, the FirstPreferenceAllotmentAlgorithm is the most reliable algorithm and should be used by default. IterativeHR is the second best algorithm, and SlotBasedHRalgorithm the third best.

The functions ‘printProgressNotification()’ and ‘printMessage()’ are used for all printing purposes, and they redirect the print to the MainWithGUI.java or MainWithoutGUI.java, depending on if the GUI was used or not. The purpose of this was so that all User Interface code could be restricted to just those 2 files. The purpose of 2 functions was that in the GUI, a progress type of notification and a generic message would be represented in the user interface in different ways.

**D5) ‘Services’ package**

This package contains the various modules that get called by the ‘ExecuteStepsForAllotment.executeAllotmentSteps()’, and each module performs a particular service. For example, all data input functions get written under the DataInput.java module. Hence, if the data input were to change from CSV files to an SQL database, we could simply modify all the functions in this 1 file and the rest of the software will still run well.

Fields, functions or files that have not been explained here should be obvious to understand from the comments in the file.

**a) CheckInputFormats.java**

This module provides 1 function for every input file, in order to check the format of each input file, so that an informative error correction message can be given to the user. If we skip this, any errors will crash the program, and show up as a ‘Input format exception in java.scanner…..’ which is difficult for an end user to understand, and also the errors will not be displayable in a GUI.

**b) DataInput.java**

All input is handled in this module. While taking in input, any unfixable errors is flagged (only the first error gets flagged), and the execution halts. An example of an unfixable error is a course which occours in the student preference list, but not on the course list. Some points to note are :

* In the populateCourses() function we create 2 versions of the course read - the inside department version, and the outside department version
* In the populateStudentPreferenceLists() function, for every course we see in a student’s preference list, we need to fetch the inside and outside department versions of the course. For example, for the preference CS1000, we have CS1000$inside and CS1000$outside. If the student is an inside department student, we add CS1000$inside followed by CS1000$outside to his preference list, and for an outside department student, we only add CS1000$outside to his preference list. Hence an inside department student will first try to secure a seat in the inside department version of the course, and only if that is full, will he try to occupy a seat in the outside department version of the course. Outside department students are restricted to the outside department version of the course only.
* In the populateStudentPreferenceLists() function, as mentioned above, we have expanded the preference list, and we have to fix the preference numbers. Hence, what we first do is that if the preference number given is X, we enter the preference number for the inside department version as (2\*X) and outside department version as (2\*X+1). Now we get a consistent preference list with each course (including each outside-inside parts of a course) having a unique preference number, but with gaps and the order in the list being that of the order in which the input was read. Next, we close the input file, and then sort the list based on preference numbers. Finally, we call the function setPreferenceNumberToPositionInPreferenceList() on each student so that the entry ‘y’ in the preference list has preference number ‘y’. We have finally arrived at a preference list which is in order and without gaps. This is important to have from the point of view of statistics calculated in the end.

**c) InputSanitization.java**

Sanitizes the input, or checks for possible fixable errors in the input. An example of a fixable error is a duplicate course. We can simply delete the duplicate entry.

Note that the use of the ’iterator.remove()’ is necessary when deleting items in the list because using arraylist.remove() will not work while you are iterating over the list.

In the checkForEmptyPreferenceList() function, we do not remove students with an empty preference list from the student list for the purpose of statistics. For example, in the mandated electives statistics, we need students with empty preference lists also, to get a correct count of the total students in the batch.

**d) GenerateCoursePreference.java**

There are 2 ways to categorize students - based on whether they are inside or outside department students, and based on whether they are high priority students or not. So this is the preference order for the students of these categories.

highPriorityAndInsideDeptStudentsInCourse

highPriorityAndOutsideDeptStudentsInCourse

lowPriorityAndOutsideDeptStudentsInCourse

lowPriorityAndInsideDeptStudentsInCourse

So, for any course, all the students in ‘highPriorityAndInsideDeptStudentsInCourse’ get ranked before the students in ‘highPriorityAndOutsideDeptStudentsInCourse’, who get ranked before ‘lowPriorityAndOutsideDeptStudentsInCourse’, who in turn get ranked before ‘lowPriorityAndInsideDeptStudentsInCourse’. Within these categories, the sorting is done based on the ranking criteria (Simple random, stratified random or cgpa).

We use the ‘Collections.shuffle()’ function for shuffling in Simple random. For stratified random, we take all the students who ranked the course as the first preference, shuffle them using ‘Collections.shuffle()’ and add them to the start of the preference list. Then we pick students who ranking the course as their second preference and repeat, and so on. For GPA criteria, we just use the ‘Collections.sort()’ function and provide a Comparator class for any 2 students which compares students based on their GPAs.

**e) CommonAlgorithmUtilities.java**

This is simply a common set of functions used by the 3 iterative alogrithms - IterativeHR, FirstPreferenceAllotment and SlotBasedHR.

**f) OutputSantization.java**

This file checks if the output given by which ever algorithm was used satisfies certain basic checks. Any error caught at this point is unfixable, and the program is halted. The problem could possibly be in the input data or the code.

**D6) ‘Tests’ package**

This package has all the testing for the SEAT software. Every time major changes are made to the software, all tests should be run once. Additionally, for every new module written, new tests should be written, and old test cases corrected as necessary.

There are 2 types of tests - Unit Tests and Overall Tests. Unit Tests are written for every function, and they test the correctness of individual functions only. These tests are written using the JUnit4 test suite and have a test for every possible testable function. Overall tests have the sample input-output pairs for the software, and only test the overall functionality of the software without going into the corner cases of each function.

The Unit tests are in the ‘tests.unitTests’ package. Overall tests are in the ‘tests.testCasesForIterativeHR’, ‘tests.testCasesForSlotbasedHR’ and ‘tests.testCasesForFirstPreferenceAllotment’. There are 3 folders because each of the algorithms will give a different expected output.

The files in this folder may not be in the exact required format because of the large number of changes to the input files and formats, and may need to be tweaked to work.

**D7) ‘SEAT-documents’ package**

This folder contains all the important documents for SEAT, including the changes to be to the software in the future, some ambititous changes, the User Guide and Code documentation.

**D8) Building the Project**

Just run ‘make’ from the folder containing the ‘Makefile’ file. It will handle the building of the project and output a SEAT.jar file. The software can then be run by issuing the command ‘java -jar SEAT.jar’.

The Manifest.mf file is needed by the Makefile to know which the main function of the application is. It should either be set to MainWithGUI or MainWithoutGUI.

Alternatively, you can import the project into an IDE like Eclipse, Netbeans or IntelliJ and it will manage the building of the project for you.