Data Structures, Collections, Generics, Algorithms and Design Techniques

Data Structures

HashMap

HashMaps are used for Companies, Owners, Students, Preferences and Teams. Using HashMaps for the storage of core objects allows low time complexity access, removal and addition with an O(1) time complexity compared with an O(n) complexity for Arrays for these operations.
Many tasks in this assignment involve getting and removing, favouring the use of HashMaps. The larger space complexity of Hashmaps compared with Arrays would be a consideration as the application scales. A side-effect of the use of HashMaps is that it does not allow for duplicate keys, this assists this implementation.

ArrayList

Temporary data structures include Arrays for sorted data, where ordered iteration is required, and an O(n) time complexity would not be avoided, because iteration is used for comparison. Further some tasks require backwards iteration (comparison for swapping) and random access, where we know the location of the selected student and team.

Stack & LinkedList

Stack was used for undo and redo functionality. LinkedList collection was used to queue suggestions from the Quadratic approach, to allow progressive delivery of time intensive suggestions in a multithreaded way, that can be progressively accessed by the user.

Collections

The HashMap Collection was used for the storage of core data, the LinkedList collection was used to queue suggestions from the Quadratic approach, to allow progressive dequeuing of suggestions after they are delivered in a multithreaded way to the LinkedList. Where iterations are required, the Collections Iterator is used, and hashmaps are converted to arrays using the toArray.
The Comparator is used to do simple sorting of teams with $O(n^2)$ time complexity, while the number of teams is small, this implementation of quicksort does not work well with low numbers of entries. When the application scales it will switch to the already implemented QuickSortTeam class for an $O(n\ log_n)$ time complexity, note quicksort is already implemented for students.

Generics

All student objects (student, studentInfo, StudentInfoWithPrefs and StudentSQL) are held in the one hashmap data structures, and cast at the appropriate time to access the functionality required. When more detail is added, the parent object is replaced with the more detailed
All Objects that represent data in the workspace are subclasses of the ParentObject class, which had the ID, and hold a static reference to the maximum objects of a class, and relevant check method.
This reduces duplication and mitigate storage growth (space complexity) that would grow at a faster rate than the rate that the application scales.

Algorithms

The balancing algorithms rely on an Objective Function outlined in Appendix A, for an $O(n\ log_n)$ time complexity and auto-balancing algorithm is used to minimise the objective function, by sorting the applicability of each student for a team, and where a student is the best for two teams comparing the overall objective function based on their assignment. For a more time-intensive comparison, swap suggestions are provided from a multi-threaded algorithm that compares each of the worst teams with best teams, and gives swap suggestions where the objective function can be minimised.

Design Techniques

Uses the singleton pattern for the central data class HashMaps, and all shared information classes are also utilising the singleton pattern to allow future removal of access from the HashMaps class.
Uses façade class for HashMaps, to allow MVC access to Controller. Additionally, a modified mediator pattern is used for swap suggestions algorithm, to disable the button as needed. Further the composite pattern is used for undo and redo as demonstrated by Charles.

Scalability

Most elements of the program are scalable, the exception being class attributes, which if they were to grow would redefinition of their accessing methods and constructors to effectively scale e.g. Number of Difficult Students (students a specified student can't work with)

Classes

The Maximum count of each object is statically defined in the ParentObject class, the maximum number of students is statically defined in the Team class. These can be can be changed in just those places once to increase the roof on new objects.

The default constructor for Team features 4 students, but an additional constructor that accepts an arraylist is also defined, allowing for scaling.

Each Preference Object is held in a Preferences class, which holds the preference in an array, however a checker method would need to be refactored to grow the preferences object.

GUI

GUI can take as many teams and students as necessary, as each team and student label and checkbox are dynamically added in the Controller from the Model.

Limitation is the Team Information dialog box is defined through scenebuilder, if the student number per team was to grow substantially (which is not thought to be realistic) additional columns can be added with relative ease, however at a point additional columns make the layout impractical, therefore an alternative layout would need to be pursued.

Reducing Duplication

Generic approach is used to reduce data duplication and complexity, where the same data structure is used as information is progressively added. For each addition of information and functionality, the class is extended, to enable reuse.

Time complexity Decisions

Data Structure

HashMaps were preferred for add, remove and retrieve operations, with a time complexity of $O(1)$ this structure does not add significantly to the running time of the application. Where sorting and random retrieval are required, arraylists are preferred due to their $O(1)$ complexity for random access, ability to retain order, and simplicity of operation.

Algorithm

$O(n\ log_n)$ sorting with linear comparison is preferred to bubble sort operations of $O(n^2)$ time complexity. As such sort operations use quicksort, apart from the current implementation of teams which has too few entries to operate effectively. However, upon scaling, this will be transferred to an already constructed quicksort algorithm.

The algorithms were built to scale, the auto-balancing algorithm leverages quicksort to enable comparisons of the most appropriate student for each team, and pick the overall lowest objective function out of this project-greedy approach. The swap suggestions, compares the students in the worst teams with the students in the best teams, starting from the worst students in the worst teams. Given the quadratic nature of this algorithm, it is prone to slowing, so multi-threading was leveraged to deliver answers where the overall objective function improves into a LinkedList, which can be polled by the user. The scalability of this approach will be further investigated.

Data Persistence

Currently the whole database is read and deleted on GUI application start, and loaded on application closure, which is an expensive operation. For multiuser environments, an alternative partial-read and update approach will be required. Furthermore, the risk of data loss is significant in the case of application failure.

Lessons Learnt

Hardcoded logic

In future I would be even more careful to avoid hardcoding functionality in any controller class, I realised late in the submission that some exceptions were being thrown by the command line menu. Luckily, these were few and non-critical, but there were further impacts on the rest of the model changing logic to effectively meet the requirements that drove these exceptions.

Map out architecture early

This lesson was the absence of a problem. I found the inheritance of classes much easier to visualise and execute by pre-mapping the classes and variables. On occasion this design thinking can go too far e.g. each personality type has its own object with a short and long name and character attribute, they are accessed through a personalities class, which holds them in an array.
This level of detail was not necessary & potentially confusing. The details could have been hard-coded in the personalities class, leaving more time to focus on other issues in the implementation.

Problem Size

I realised a large problem is many times more difficult to solve than many small problems. My swap suggestion algorithm was extremely challenging to implement until I recognised that I was attempting to solve several smaller problems simultaneously.

Singleton Pattern

Passing data between controllers was a very challenging problem to solve, it took several hours of searching a incremental build to create a workable solution. If I had known at that stage about the singleton pattern I would have solved the problem much more quickly. However, in the process of investigating this problem I gained more of an appreciation of the JavaFx approach.

Focussed Classes and Methods

Where my classes were more focussed, they were easy to reuse and understand. The process of extending an SQL interface was not much of a challenge due to the focussed classes I was extending.
Unfortunately, my façade class at various times grew significantly in size, this made navigation through the class very challenging, and any reuse of methods a longer process. By improving my use of generics, and passing processing where appropriate to other classes, I slowly shrunk the code. Other SOLID principles immediately made sense when introduced, I therefore further refactored my code with these in mind and will start with these in mind in future.

Refactoring is rarely a big bang

As I noticed my façade class bloat to the point of difficulty, I was determined to break the code into 3 classes, two new for processing and the existing one for access. A few hours turned into an all-nighter, and by the morning my program was broken beyond repair. I underestimated the complexity of a big bang refactoring approach. After throwing all my changes out, I realised I had learnt two things: first, architectural changes during delivery are extremely hard to execute and secondly that small, deliberate refactoring efforts are much more effective than large ones, as there is too much to keep track of. It seems the harder you push a program to bend to your refactoring will, the more it will push back against you.

Consider scale

My use of scenebuilder meant that I was drawn to using static elements beyond their usefulness e.g. for each of the student and team labels and checkboxes. The static if-then coding, was a significant initial effort, I spent far less time dynamically declaring labels and checkboxes, for an overall superior solution. So, consider the potential scale of your application, and if you are manually defining each instance, is each really separate in their own right?

Threading and acting on the JavaFx Thread

I found acting on the JavaFx thread a difficult concept to understand, however my significant investigation helped me learn something new and fill gaps I had in my understanding of threads.

Appendix A: Algorithms

Objective Function

The objective function will not reduce standard deviation, but focussed on improving each metric, and prioritises based on User Defined Weightings. While it has the general effect of improving

Default Weightings

The default weightings are 60% for skill shortfall, 30% average competency, and finally 10% for first and second preference.

The reasoning for this weighting is that good projects are central to the success of this assignment, if companies who provide quality projects are rewarded, this will help future semesters. Particularly if the process of shortlisting is discouraging to company participation. If a company doesn't have any projects left, they might decide not to put forward projects, or put effort into projects.

The average competency is important, but a low overall skill shortfall has the side-effect of reducing the competency gap by balancing students who are not academically gifted into teams that are competent.

Finally, the first and second student preferences are not prioritised because the projects that remain have been shortlisted, they are quality projects, so it does not necessarily follow that students should be disappointed by the remaining projects. Particularly because in general, students do better in subjects that they like, so they are likely to be matched with appropriate projects through the skill shortfall approach.

Calculation

The objective function for a student is a count of the skill shortfall * shortfall weighting + variance from average competency * variance weighting + where a first and second preference is not given * weighting for preference.

Auto-Balancing Algorithm

No hard rules are violated in the two approaches. each comparison is checked so

Find Each team and their project attributes

This approach loops through every team, and finds their project attributes that are relevant to the objective function.

Sort Leader Students

To avoid the "NoLeaderException", a leader is selected first for each team. Based on the objective function calculation, leader students are sorted, using a quicksort by best (min) to worst (max) per team. The best student for each team is then looped through until each team has a student assigned. In cases where the student is the best for more than one team, the existing and new objective function are compared and the better function is chosen.

Given only one student has been selected, the additional exceptions do not feature yet.

Sort the remainder of the students

All remaining students are sorted in the same way, and the same process is used to assign students, however, there are additional helper functions that will skip students who will cause a personality imbalance or student conflict.

Once all students are assigned the suggestion is provided in a dialogue box, with the option to apply or cancel.

Swap Suggestion Algorithm

For a more detailed, but higher time complexity approach, each already assigned student in the lowest performing teams, starting with the lowest performing student is tested against the teams in the upper half of performance, starting with the highest performing students.

To mitigate the potentially large amount of time this would take, each lower student comparison is launched in a new thread, with only a linear loop $O(n)$ occurring on each thread.

If the swap improves the aggregate objective function and does not violate any exceptions, the suggestion is added to a LinkedList Queue. On the first LinkedList entry the button to dequeue and view the suggestion is enabled.

On change actions. E.g. removing a student or swapping students, all running threads are cancelled, and the queue cleared.

On ready actions e.g. loading the view, adding a student, or swapping a student, the swap suggestion algorithm is relaunched to analyse the current set of students.

Appendix B: Design Patterns

Singleton

Command Manager – to allow consistent access to the undo/redo functions, the reference to the stack and command logger and persistent as they are initialised as class variables.

N.B. Controllers do not follow this pattern as they access a reference from the appropriate javafx file.

Below are other singleston classes:

- HashMaps façade class
- DefaultObjectiveWeighting
- AverageGrade
- Count
- StandardDeviationWeightings

Façade Pattern

Façade Pattern is used for the HashMaps class, allowing simpler MVC implementation

Mediator Pattern

Modified Mediator pattern is used for the swap suggestions algorithm to ensure the suggestions button is disabled when a change to teams has occurred, being available when the new suggestions are available. The queue is cleared by the Controller, so does not feature in this pattern.

Composite Pattern

Composite pattern for undo and redo, as demonstrated by Charles.