Design and OOP

This program utilised data structures which can dynamically expand to account for different numbers of companies, students, projects and owners. I primarily used HashMaps for theses as each of the objects have a unique identifier. HashMaps are easier to obtain the unique objects via their key and also, it's easier to update the objects if they have the same key. Also, I've used ArrayLists in some classes like the GUI controller to populate the list view of projects. I've also used HashSet to ensure that the user can only a number between 1-4 to rank skills for each project once. HashMaps, HashSet and ArrayList are dynamic and can extend depending on the change in number of projects, students, companies and owners. In addition, I've used generics in my parameterised Stack class. This way different objects can be stored in each stack should the program need to hold different types of objects and keep a record of actions.

This program is primarily built on OOP principles and the mediator design pattern. In this program the MenuOptions class. This is the main class which is the one only aware of the existence of other classes. The other classes like Projects and Companies, are not directly aware of all the other classes. It is also the one which stores all the projects, students, companies and owners. Other classes like the ProjectTeamFormationController (GUI controller) actually have to obtain information from the MenuOptions class and not directly from the model Student class. The mediator design pattern enables some separation between low level model classes and other functional classes.

The stack class uses the singleton class, which enables the class to only make one instance of it. This way the program will not use any of the other stack and all actions are stored in a single data structure.

This program uses a number of utility classes such as WriteMethods, StandardDeviation, SortAlgorithm, which can be used in multiple places and methods, in the GUI, in the MenuOptions class. This enables code reuse.

This program also demonstrates a number of the SOLID design principles. This program demonstrates the single responsibility principle for most classes. For example, methods and attributes of the Student class are unique to the student class. Also, this program uses a lot of composition to access utility methods in a number of utility classes. In addition, this program follows the Liskov Substitution principle. Since using inheritance would break the Liskov substitution principle, I have instead used composition. This way there is nowhere a class inherits not needed methods.

The GUI design also uses the MVC design. It separates the model from the view. Considering that in industry the UI changes more often than the business logic, the changes are mediated from the controller.