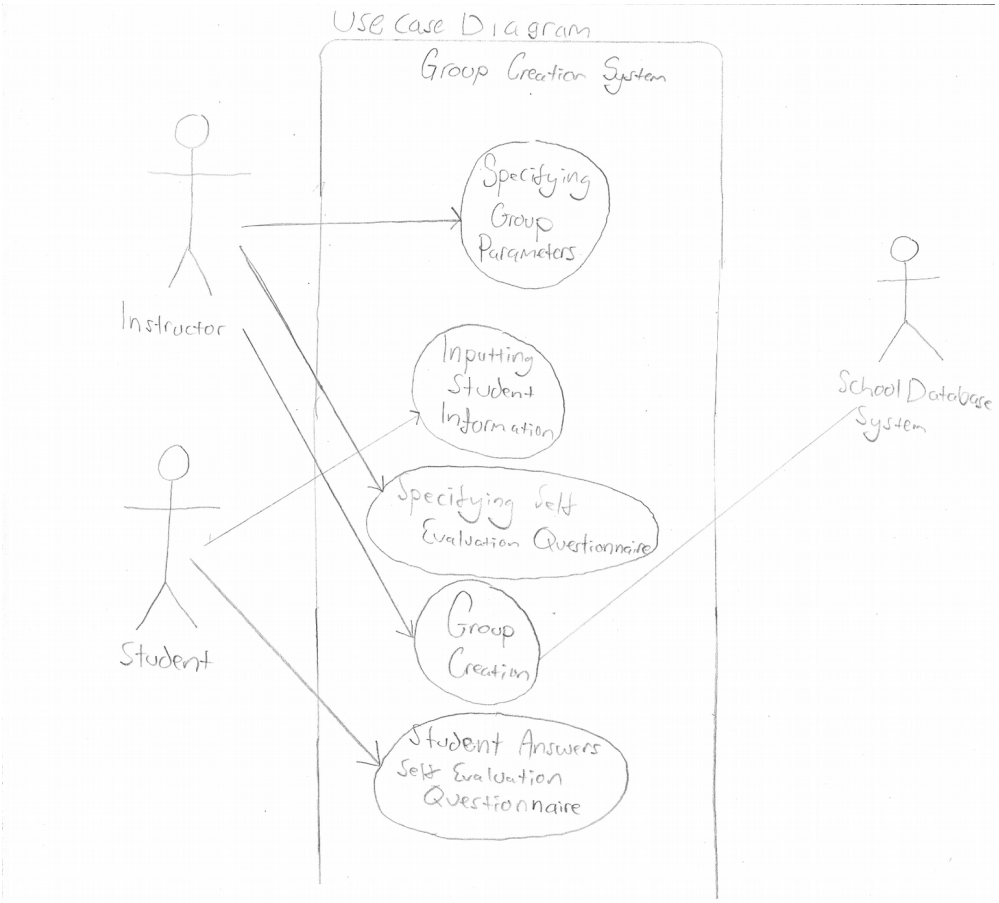
Iteration 2. Design

Overview: In iteration 2, we will focus on allowing the instructor to edit and specify his preferences for groups. As well, we aim to allow groups to be sorted by students GPA which will be retrieved from the system. We show how this should proceed in the following use cases, which correspond to the use case diagram seen below. We first develop design for Specifying group parameters and generating groups as seen in the use case diagram below in full. This is involves updating our design from iteration 1, in order to evolve with our software. As we focused on the alternate path of allowing the instructor to edit the use case in the previous iteration, in this iteration we tackle the main path, including introducing an algorithm to handle group creation with instructor preferences. To do this we make use of the strategy pattern in order to ensure cohesion in our code and the correct decoupling between classes.



Use Case 1,
Specifying Group Parameters

Actor: The instructor

Description: The instructor creates a new set of groups based on a list of students and determines the maximum group size, specifies which students can and cannot work together, and whether or not the groups are to be created based on skill. He or she must also specify a deadline for students inputting personal information into the system. If the instructor wants groups to be created based on skill, a self-evaluation questionnaire must be provided. All the parameters input during this step can be edited until the instructor decides to continue with group creation, at which point they are no longer editable. The parameters, once entered, are available to the students.

Flow: See the following page. The first alternate path addresses the case where the instructor wants to create skill-based groups. The second alternate path shows the flow in the case where the instructor is editing parameters that were made previously.

Preconditions:The professor must be registered at the institution and teaching a course during the current term

Postconditions: Maximum group size, whether or not the groups are skill based, and a deadline must be specified. Students are notified about the information requirements and deadline

Use Case 1,
Specifying Group Parameters

Main Path	Alternate Path	Alternate Path 2
1. instructor starts a new set of groups		1.2 instructor opts to edit a set of groups
2. The system retrieves the list of students from school database		2.2 instructor edits the list of students
3. instructor specifies the maximum number of students per group		3.2 instructor edits the number of students per group
4. instructor specifies a deadline for accepting student information		4.2 instructor edits the deadline for accepting student information
5. instructor indicates that groups do not need to be based on skill		5.2 instructor edits whether or not groups are created based on skill
	5.1.1 instructor indicates that groups are based on skill	5.2.2 instructor edits self evaluation questionnaire
6. instructor specifies what students can and cannot work together	5.1.2 instructor specifies self evaluation questionnaire	
7. Information about group parameters is made available to students		

Justification: In order to create groups, the size of the groups must be input, and as creating groups is the point of our software, inputting group parameters is essential to our system. Therefore it is important to analyze all possible paths which may be needed in order to accomplish this in a manner that will be satisfactory to the user of this software. Therefore a use case diagram seemed like the natural approach.

Use Case 2,
Group Creation

Actor: The instructor

Description: The instructor initiates the group creation process. The groups are created based on the group parameters entered previously, and the instructor has the ability to modify groups. If modified groups violate the group parameters, then a warning is displayed. A warning is also displayed if the system is unable to create compatible groups.

Flow: See the following page. The first alternate path addresses the case where the instructor wants to create skill-based groups. The second alternate path shows the flow in the case where the created groups violate the original group parameters.

Preconditions: Group parameters exist, all students have completed the self-evaluation questionnaire (if required) and have provided extra-curricular availability information. The deadline for student information being submitted has passed.

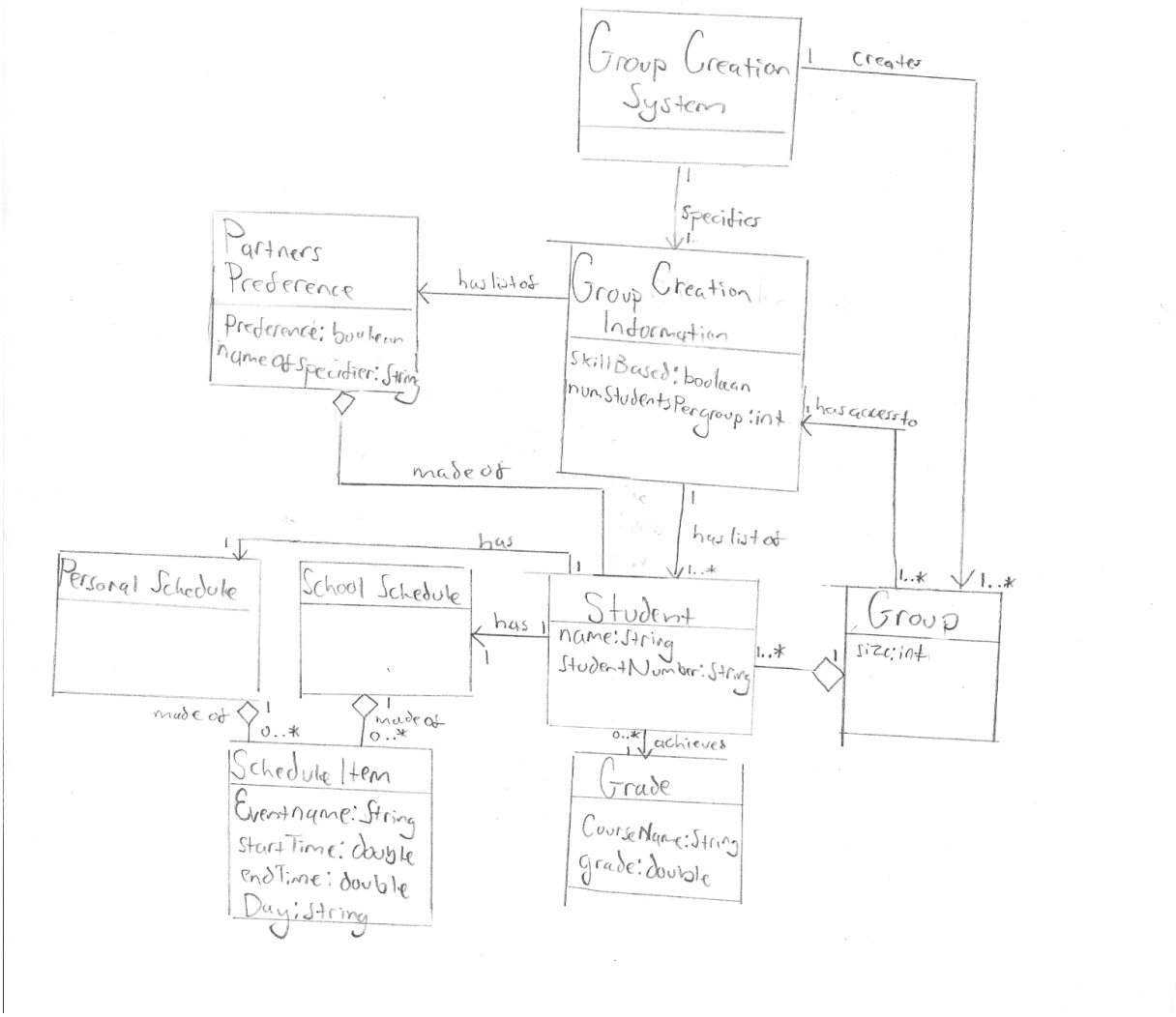
Postconditions: Groups have been created, no groups exceed the maximum group size, if groups violate initial group parameters the instructor has acknowledged the warning, groups are available to students.

Use Case 2,
Group Creation

Main Path	Alternate Path	Alternate Path 2
1. instructor chooses to start generating groups		
2. System retrieves class schedule information for each student from the institution's database		
3. System determines the number of groups and the number of students per group		
4. System generates groups based on instructor input, available meeting time, and student preferences (in that order)	4.1 System generates groups based on instructor input, skill balance, available meeting time, and student preferences (in that order)	
		5. System is unable to generate groups with sufficient meeting time. A warning is displayed to the user.
6. instructor reviews and edits generated groups		6.1 Edited groups violate some of the initial criteria. A warning is displayed to the user. Repeat step 6 until instructor decides to confirm.
7. instructor finalizes groups		
8. Groups are made available to students		

Justification: This is the most important feature of our software, as seen in the feature list above, and therefore of utmost importance. As well, it does not appear straightforward to implement. Therefore this satisfies all three of the three Q's of architecture, and it is essential to handle this feature first in order to reduce risk. Therefore we aim to analyze group creation in as many different ways as we can in order to fully understand the core feature of our software.

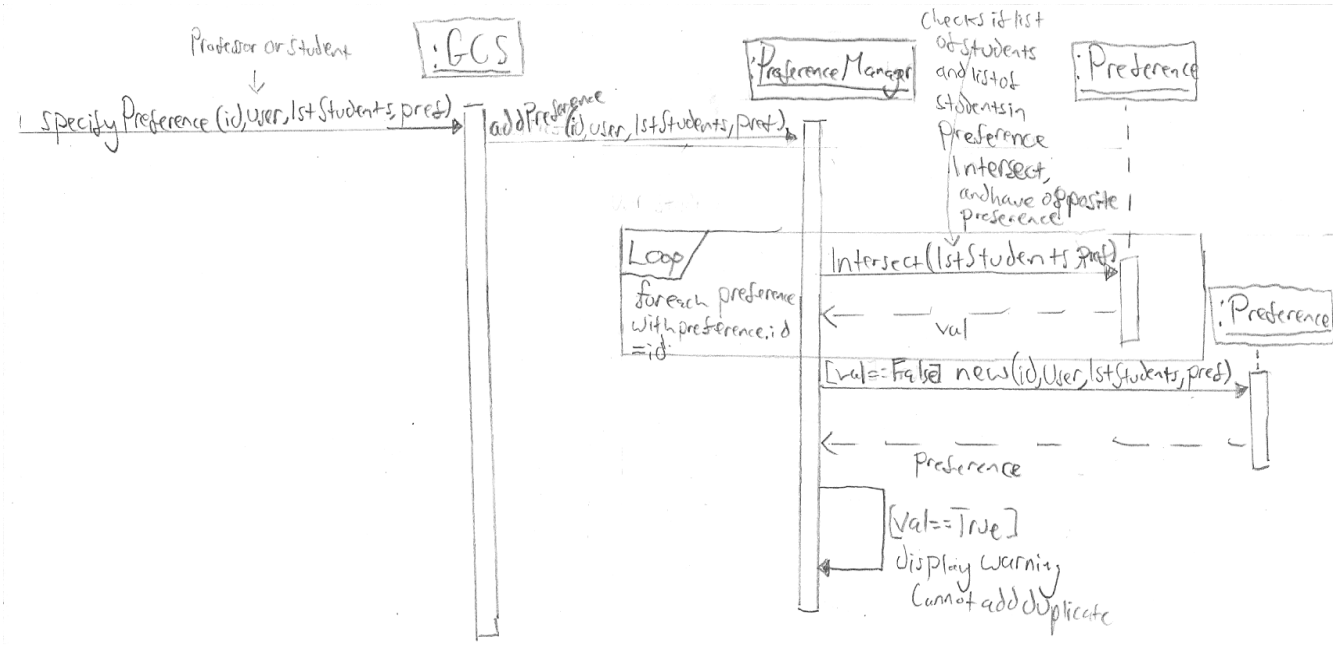
Domain Model



We include the domain model from the design of the previous model, as this is still useful, giving us a simplified overview of theoretical classes and their relations to one another. This allows us to plan for the future by observing how in the most simplified form, these classes interact. This planning will allow us to create design which is reusable, modifiable and evolves as our software changes.

Sequence Diagrams

Sequence Diagram 1.
Specify Preferences



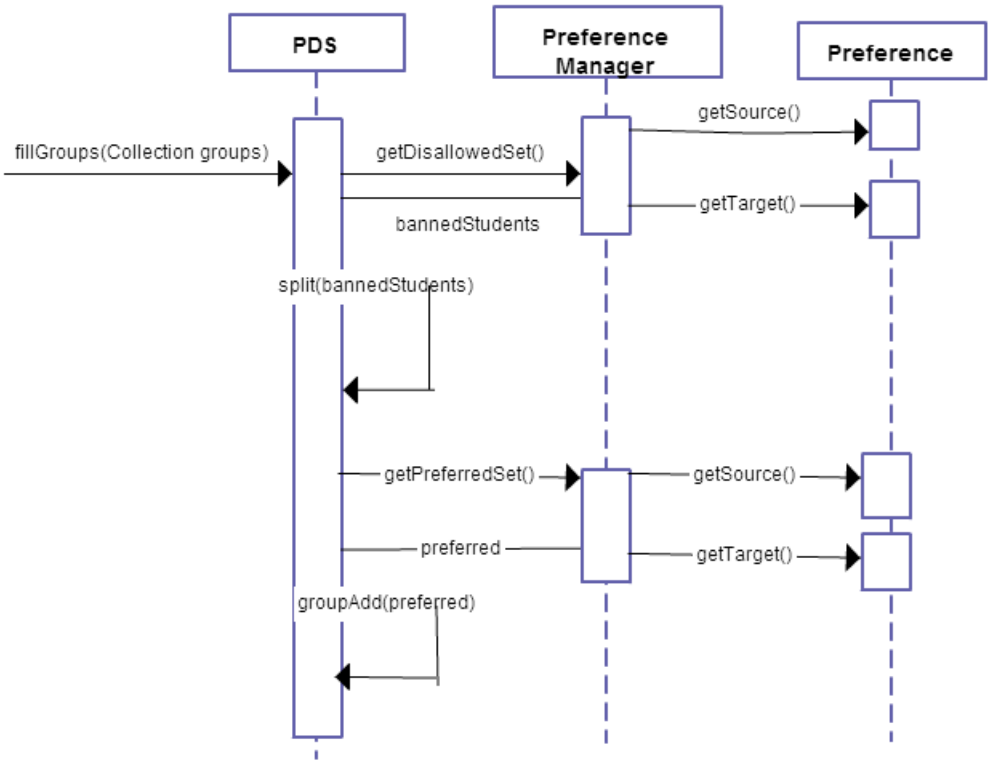
This sequence diagram shows the process of specifying desired (or undesired) groupings. An important feature of our system as seen in the feature diagram is the ability to specify preferences of groupings. It is also a piece of our architecture that a lot of our system relies on, and therefore to not consider this early on would increase risk. To reduce risk, we decided that modelling this this process with a sequence diagram would give us a very thorough understanding of how the system would work. Therefore this falls under the category of question 1 and question 3 of the three Q's of architecture. We strived for reusable design, as this design can be used for both the instructor and student to specify their preferences. The id is used for security purposes and for the ability to ascertain exactly who submitted the preference, while the user can be either 'student' or 'instructor' and can be used to determine whose preference they are for. Of course in our implementation, we will only be using instructor preferences.

We believed that this sequence diagram would be the best place for the system to check if there exists any conflicts between this users previously submitted preferences, (for example if they had previously said that two people should work together, but are now saying that they do not want those people to work together), and if so send a warning and not allow the preference to be added. To do this we define an 'intersect' method to see if the students in 'lststudents' intersect any of the preferences stored in the preference manager with the same user id as the current users id. If so they reject it and send a warning. The user may then edit their submitted preferences if they want to change their preferences, but this is not covered in this use case. The preference class has the intersect method due to the information expert pattern, as the preference class has the most information about what students it contains, its preference, and the id of the person who submitted it, therefore it is the information expert.

As stated previously we created a preference manager to ensure that each class has exactly one responsibility. The preference manager is the unique class which handles preferences. We believed that rather than overwriting, or adding any conflicting preferences it was best to just reject the addition of the preference and warn the user. This is because we believed that any other option would cause for a confusing design, and therefore the best option was just to let the user edit his previous preference if he wishes to add this one.

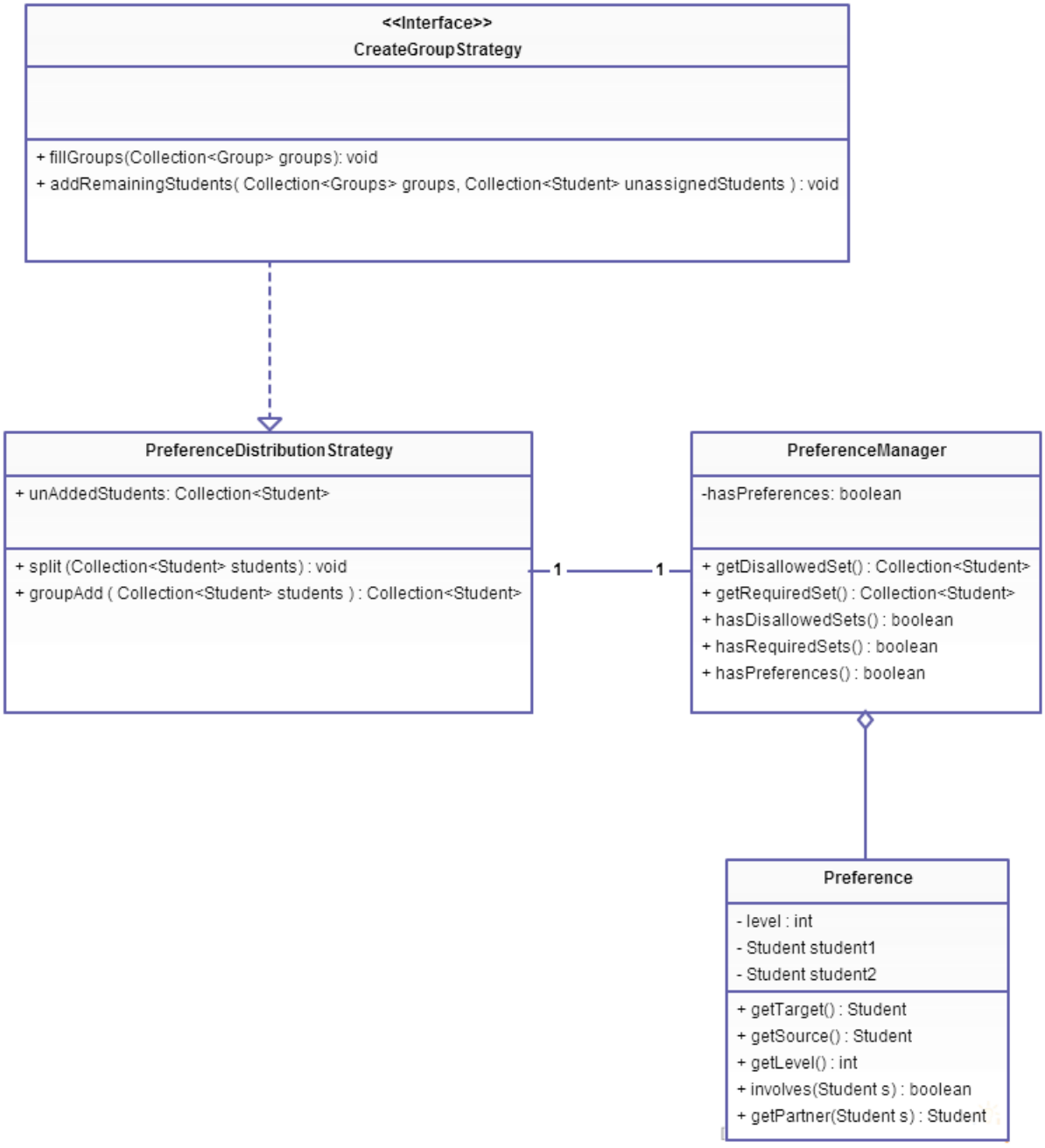
Sequence Diagram 2. Fill Groups

As we move on to iteration 2, we will be tackling new aspects of the problem. Therefore, to plan for how to implement these, we will create sequence diagrams in order to determine the correct way things should be implemented. We strive for re-usability and decoupling by implementing the strategy pattern in our design by creating an interface, group creation strategy, which will be subclassed with instances of the strategy to create groups. The preferenceDistributionStrategy(PDS) class is one such instance. This is an instance of the open closed principle, as we are closing the groupCreationStrategy to modification, and opening it to extension.



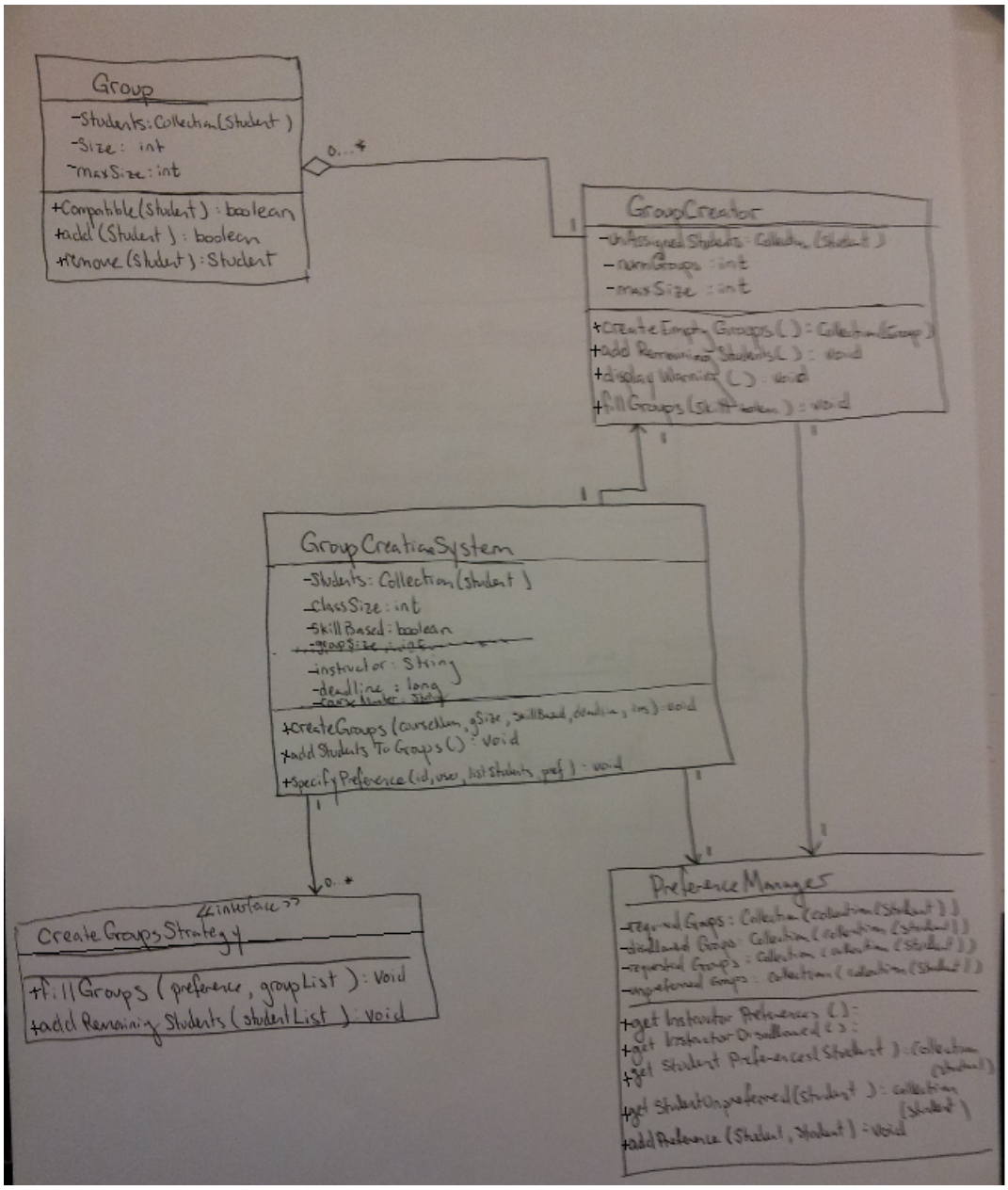
We create this sequence diagram as filling the groups is essential to our system, is difficult to understand, and we are not fully sure of how it should be done. This satisfies all three questions of the three Q's of architecture and therefore is extremely risky. Thus, we tackle this with all the design we can. We apply design patterns and OO principles as much as possible in order to ensure limit dependencies, as we believe this software will become large, and therefore it is important to keep the seperate parts of our system compartmentalized.

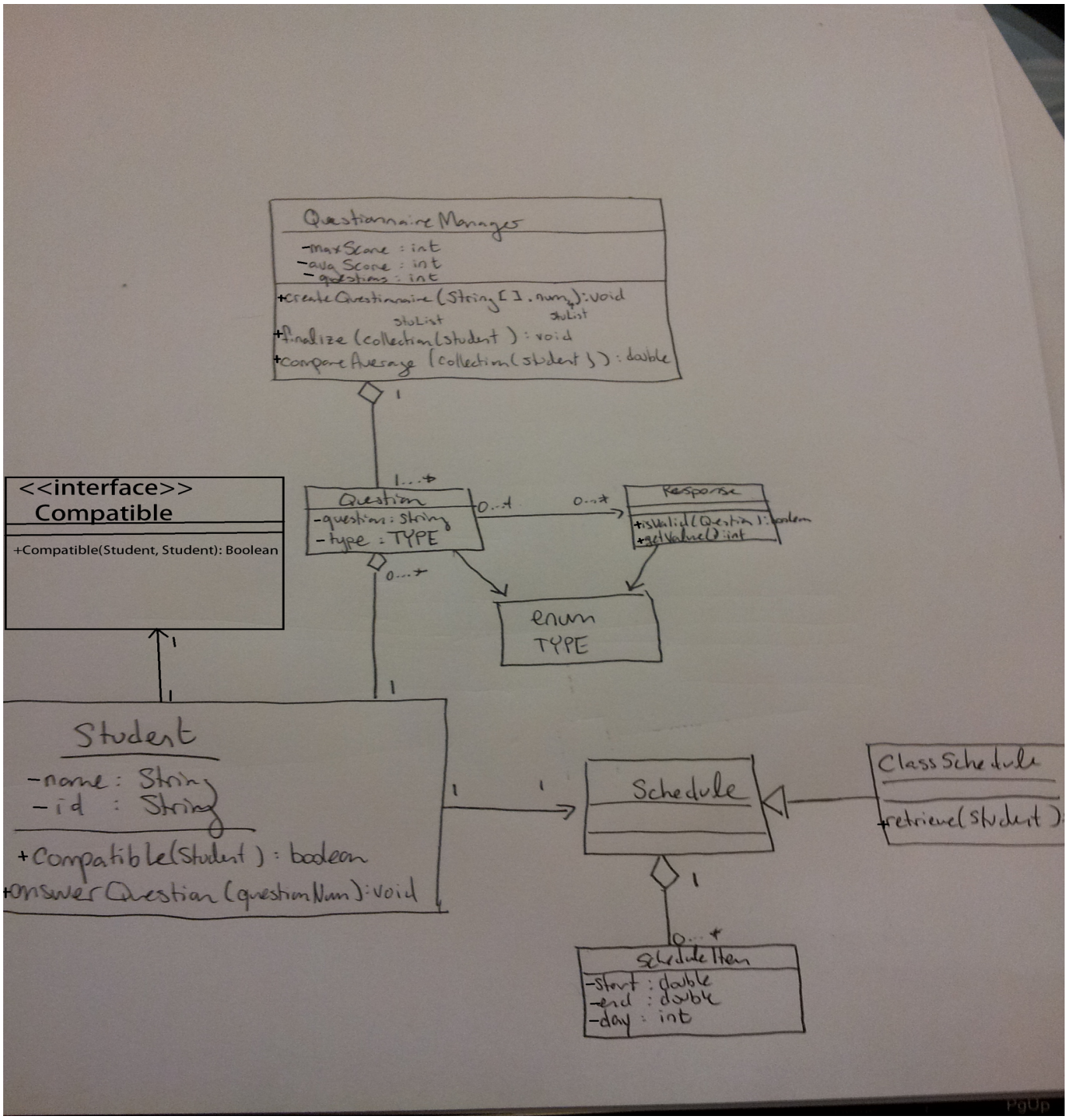
Class Diagram



We create the above class diagram in order to structure our system and design the modules we will require. The above diagram focuses only on the interactions between the preferences and the create group strategy. We do this in order to design our code in a decoupled fashion, ensuring that we

minimize dependency in every way possible. The strategy pattern can be clearly seen in the above diagram as group creation strategy. As well, include the domain models from the previous iterations which we will build this iteration on.





We have omitted from our design classes which are purely done in implementation and do not require significant software design such as “dummy” classes for the model view controller patter which we implement, and classes which handle the GUI. These are outlined in the modules as we believe it is important to understand where in the software these should exist, and what dependencies between them should and should not be present.

Modules

We have broken up the project into modules in order to work on it as efficiently as possible. Modules were chosen based on the amount of dependencies between the classes for the project, with the goal being that each module could be worked on separately and would represent the various features of the system. We designed our models to strive for a design which was as decoupled as possible. Since we were able to break up our classes from the Class diagram into two (mostly independent) diagrams, we used these diagrams as a basis for how we chose to separate them into modules. We have made sure to apply the model view controller pattern in order to separate functionality, state, and GUI from one another. This decreases the risk of our software implementation as dependency is greatly decreased, and cohesion is increased.

Utility: The utility module will consist mostly of classes which act as data structures for the more elaborate classes in the other modules. We have also placed the Registrar class in this module temporarily, as the details of how our system communicates with the school database are still unclear. We placed the Compatible interface here as it belongs with the Student class, as it is used to determine similarity between students.

Classes: Registrar, Group, Student, Compatible (interface), Controller, StudentFileReader, StudentReader(interface)

Group: Classes relating to basic group creation will go here. These classes form the basis of our system, so it seemed logical to place them all in the same module. We have already built some of the methods in these classes in order to ensure that our system is on its way to becoming fully functional.

Classes: GroupCreationSystem, GroupCreator, PreferenceManager, createGroupStrategy (interface)

GUI: Holds the classes related to user interaction and graphical display. These will be entirely for recieving and passing on information and displaying information to the user. The controller will handle any sort of action and processing data. The view class oversees the groupProjectGui and sends information to the controller.

Classes:GroupProjectGui, View