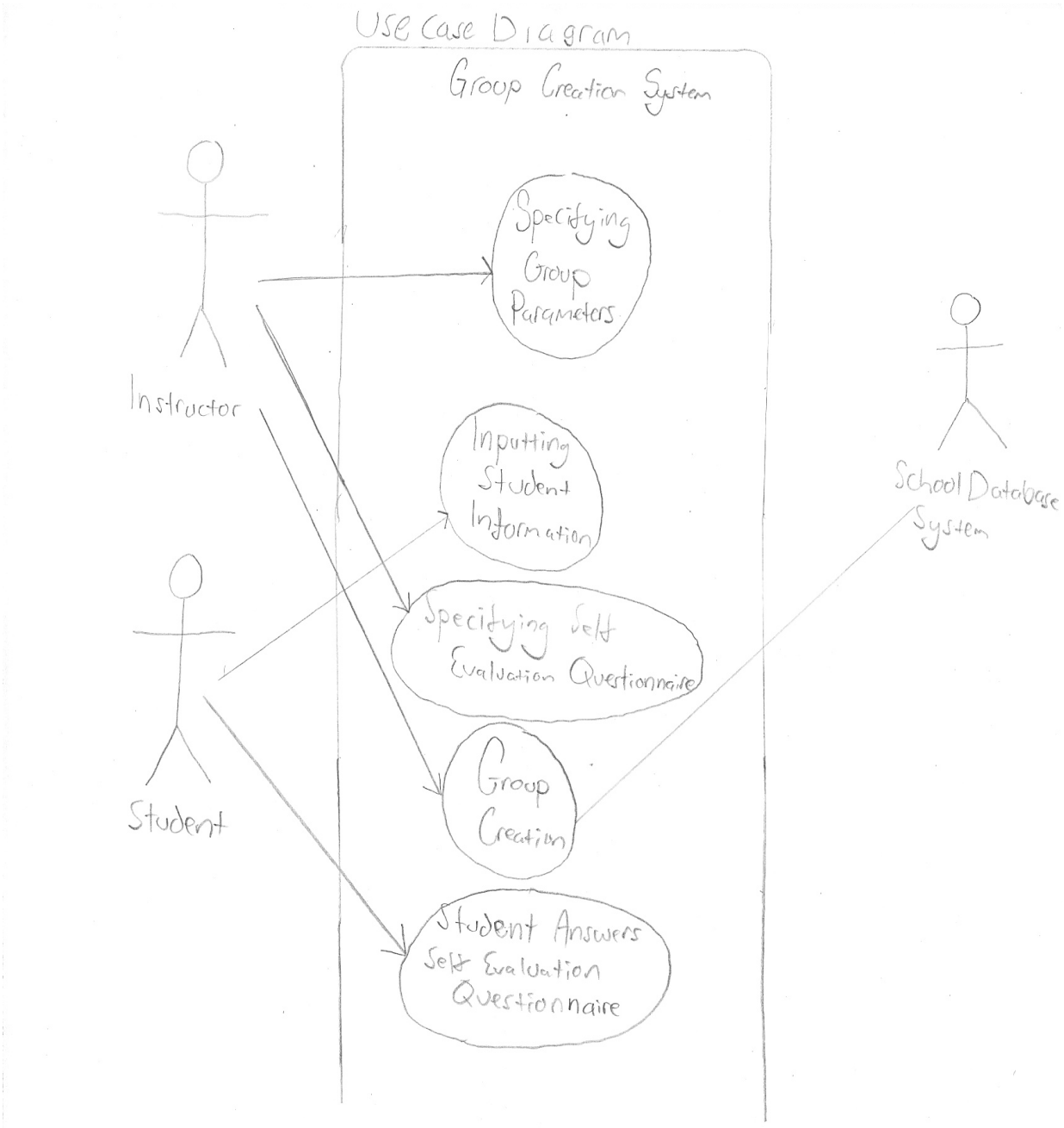
3716

Project Design

Use Case Diagram:



Use Cases:

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,
Inputting Student Information

Actor: A student

Description: The student is able to view parameters input by the instructor in Specifying Group Parameters use case. The student provides personal information such as availability outside of class hours, and preferences for working with other students. If the self-evaluation questionnaire is available, it must be filled out. This information may be edited up until the deadline provided by the instructor.

Flow: See the following page. The first alternate path addresses editing of student information. The second alternate path handles if a questionnaire exists for the student to answer.

Preconditions: Group creation parameters exist. The student is registered as a student of this course, and the deadline for submission of information has not passed.

Postconditions: Extra-curricular schedule information and answers to the self-evaluation questionnaire (if required) must be presented.

Use Case 2,
Inputting Student Information

Main Path	Alternate Path 1	Alternate Path 2
1. Student views information made available by the professor, including deadline, and the information the student must submit.		
2. Student specifies which students he or she does or does not want to work with	2.1 Student edits which students he or she does or does not want to work with	
3. Student enters times during which they are unavailable for group meetings	3.1 Student edits times during which they are unavailable for group meetings	
	4.1 Student edits the self-evaluation questionnaire (if available)	4.2 Student answers the self-evaluation questionnaire (if available)
5. Student submits the information.		

Justification: As group creation must take into account student preferences and schedules, we believed this to be an essential feature of our software. As there are many paths to accomplish this in a successful and user friendly manner, we believed that a usecase was the correct approach to this, in order to ensure that we have captured every path that a user might want to take through this part of the software.

Use Case 3,
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 3,
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		
	4.1 System generates groups based on instructor input, skill balance, available meeting time, and student preferences (in that order)	
4. System generates groups based on instructor input, 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.

Use Case 4
Specifying Self Evaluation Questionnaire

Actor: The instructor

Description: The instructor is able to specify a questionnaire for the students to answer. The instructor specifies the number of questions in the questionnaire. The instructor will specify the question and the type of question which it is. The instructor is able to edit the questions and the number of questions before submitting it. Once submitted, the questionnaire will become available to the students to answer.

Flow: See the following page. The main path addresses the instructor creating a questionnaire, while the first alternate path handles if the professor changed their mind about having a questionnaire. The second alternate path addresses the professor editing the questionnaire before submission.

Preconditions: The instructor has specified that group creation will be skill-based.

Postconditions: A questionnaire has been created and students have access to the questionnaire.

Use Case 4
Specifying Self Evaluation Questionnaire

Main Path

1. Instructor chooses to provide a questionnaire.

2. Instructor decides on the number of questions in the questionnaire.

3. For each question the instructor specifies the type of question.

4. Instructor writes each question.

5. Questionnaire is confirmed with the professor.

6. The system verifies that the questionnaire is complete and that there are the correct number of questions in the questionnaire.

7. Questionnaire is submitted to system and made available to students.

Alternate Path 1

1.2 Instructor edits whether to provide a questionnaire.

Alternate Path 2

2.3 Instructor edits number of questions on questionnaire.

3.3 For each question the instructor edits the type of question.
4.3 Instructor edits each question.

Justification: We decided that we needed to make some choices related to the questionnaire in order to fully understand what we need to do when it comes time to implement it. This reduces risk for our architecture, as group creation may rely on a questionnaire. Therefore understanding the questionnaire is essential for having our software work correctly. When evaluating the three Q's of architecture, we realized that we did not entirely understand what the user would want implemented in this feautre, and therefore ignoring this feature would be risky. Thus, we believed that exploring it through a usecase would shed some light on how this feature works and would decrease the risk.

Use Case 5
Student Answers Self Evaluation Questionnaire

Actor: A student

Description: The student fills out a questionnaire provided by the instructor if one exists. The student will answer each question in the questionnaire and the system verifies that the questions are answered correctly. The student may edit their answers to the questionnaire as long as it is before the deadline.

Flow: See below. In the main path the student accesses and fills out the questionnaire which is verified by the system. In alternate path 1, the student edits their answers to the questions in the questionnaire.

Preconditions: The instructor has submitted a questionnaire to the students. The deadline for completing the questionnaire has not passed.

Postconditions: An answered questionnaire for this student has been submitted to the system

Use Case 5
Student Answers Self Evaluation Questionnaire

Main Path

1. Student accesses and views questionnaire

2. For each question in the questionnaire student submits an answer

3. System checks to ensure that all questions are answered, and in a valid way. The comment will be specified as valid or invalid according to some strategy.

4.Student submits completed questionnaire to the system

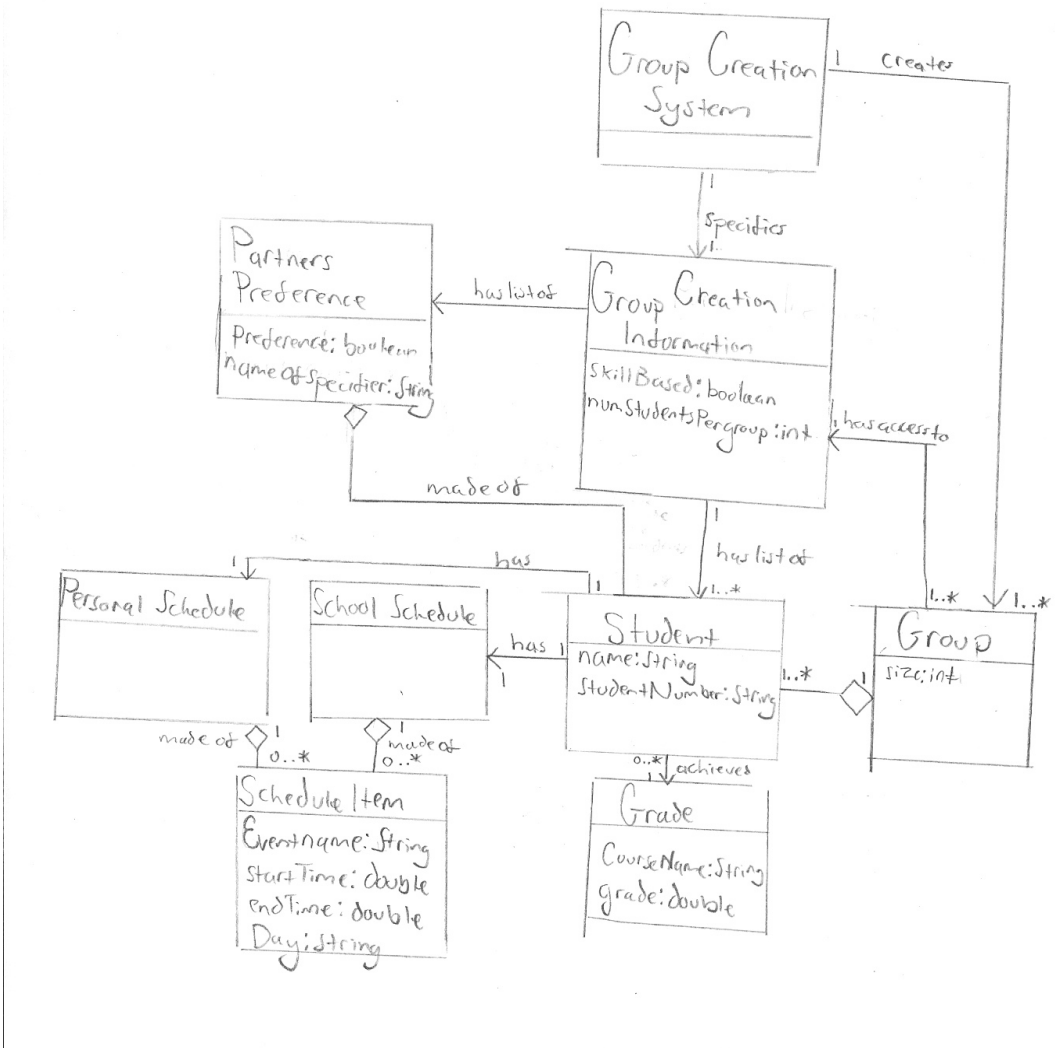
Alternate Path 1

2.2 Student edits their answers to each questions in the questionnaire

Justification: For the same reason as stated above, we believed that this was not entirely intuitive, and since group creation may rely on having a completed questionnaire input, examining this would be a way to reduce risk. A usecase is one of the simplest and most logical approaches to examining how a piece of software should work, so we believed that this would be enough to understand how the student should interact with the questionnaire.

Domain Model

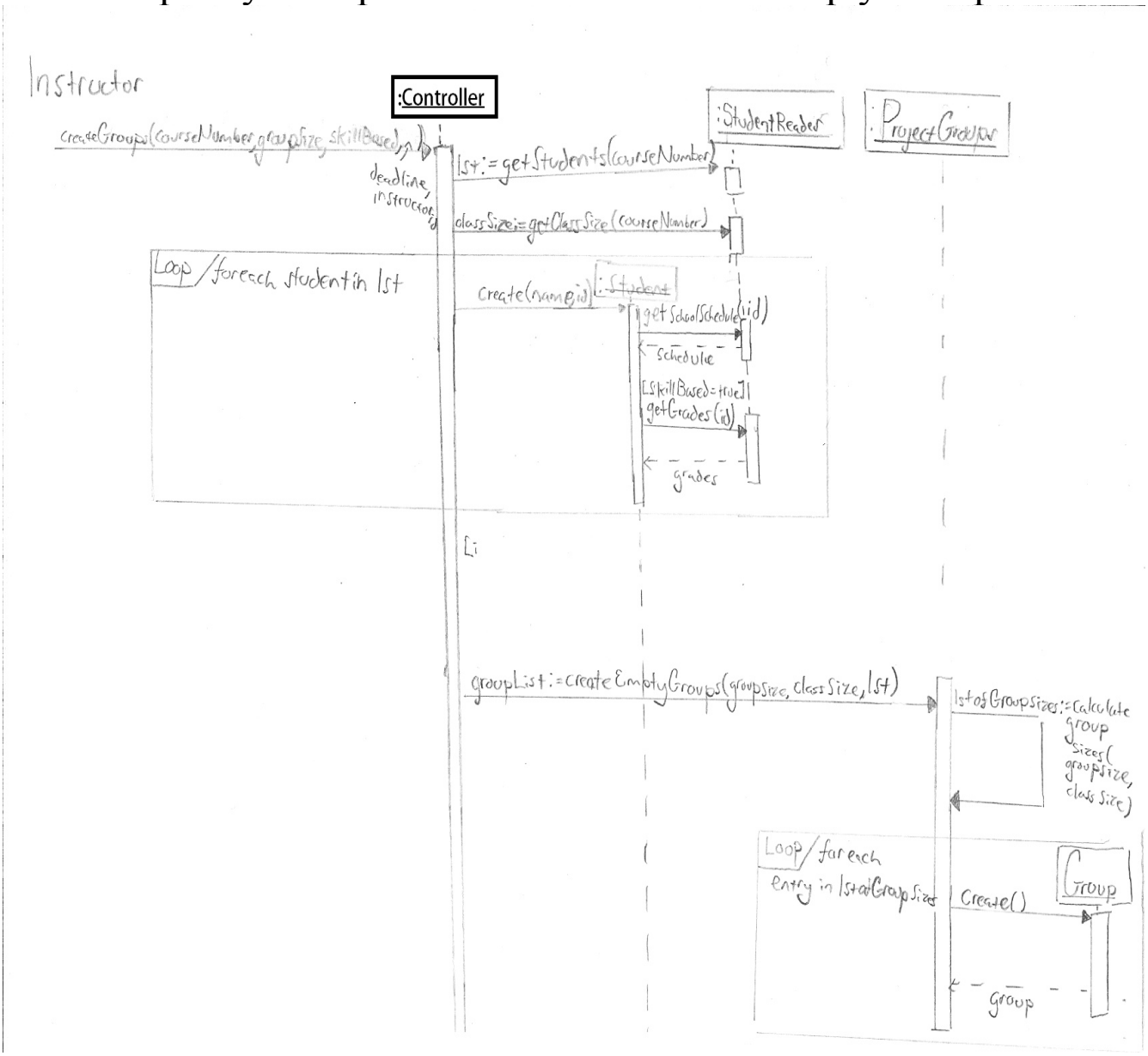
Domain Model



We include a domain model for the design, as it gives 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 Group Parameters and Create Empty Groups



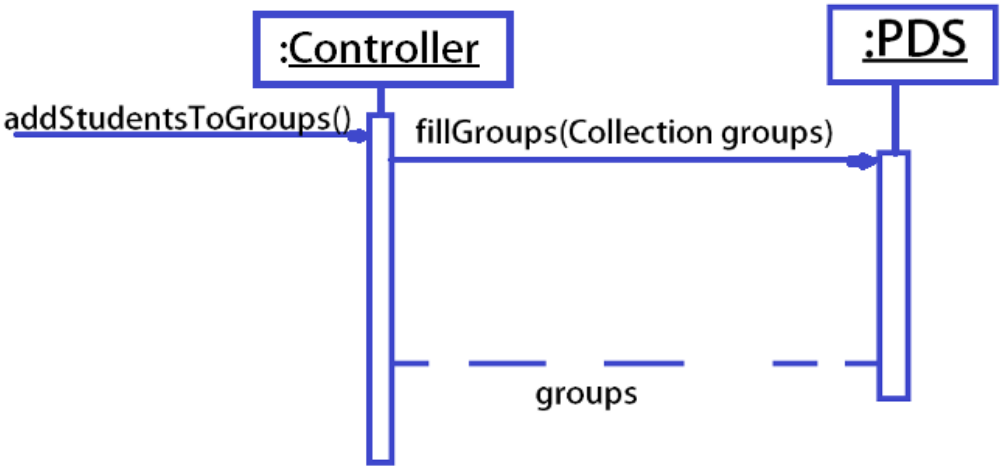
The above sequence diagram shows the sequence which occurs when the group parameters have been entered into the system. The system prepares for group creation by creating an appropriate number of empty groups and fetching student information from the Registrar. We believed that this was one of the most important sequence diagrams as it lays the groundwork for how the system will act from then on. Therefore it is part of the essence of the system. As well, we say many drastically different ways in which this process could have been implemented, and believed that creating a clear and objective

sequence of events would clear up any sort of confusion on how we intend our architecture to implement this process. This sequence falls under two of the elements of the three questions of architecture, 1. It is part of the essence of the system, and 2. We have a lack of understanding on how it will be implemented exactly. Therefore we believed that it was very important to discuss in detail.

We created a Controller class which delegates information and user input to classes to handle. This was done to ensure that each class had exactly one purpose and ensure cohesion and decoupling of our code. The registrar is an outside system which we will not implement as a class, it is simply the registrar of the university which we retrieve information such as grades, class schedule, and student list from. Therefore it is important to handle. We believed that the simplest and most user friendly way was to have the instructor input the class number and have the system query the registrar for the list of students. This greatly decreases the amount of work the instructor must do, rather than having to input the class list personally, or import it, and increases user friendliness of our system.

A design choice of ours which I believe drastically reduces work in sequence diagram 2 and 3, and avoids errors, is the choice to generate empty groups in this step. We chose this to ensure that when we had to populate the groups, we would already have the correct number of groups and sizes of each group, which avoids us the trouble of later having to create the groups before populating this. This would also allow us to partially populate some of the groups at some time if the user wanted to do that.

Sequence Diagram 2.
Fill Groups with Students



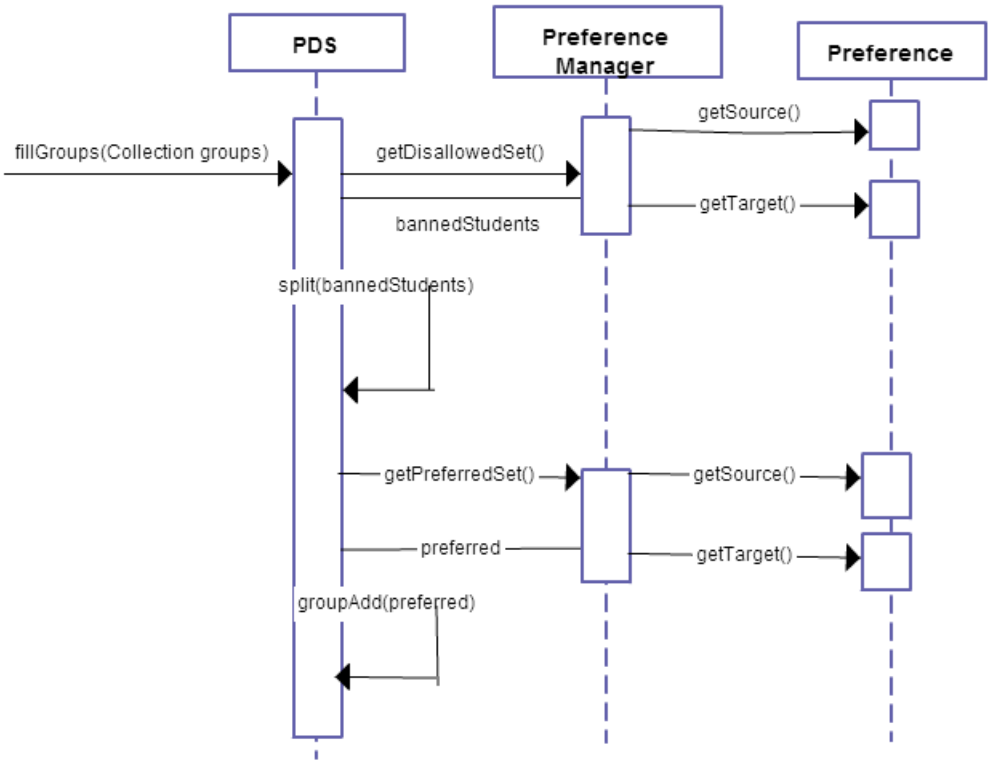
The above sequence diagram displays how the system would fill the empty groups of students created in sequence diagram 1. The fillGroups method is detailed in sequence diagram 3. We chose to do this sequence diagram although it is very simple because it is essential to our architecture. Most importantly, it helps show the flow of messages our system goes through when performing its processes.

We are designing a system for splitting students into groups, therefore it is the most important feature of our system as seen in our feature list. As well, there are many ways to accomplish this, and it proves

to be the most difficult to plan for and decide how to accomplish piece of software in our system, and therefore prioritizing it by understanding its design early on reduces risk for our software design significantly. PDS stands for preferenceDistributionStrategy, which is an interface. We use this in order to allow people to specify how the groups will be created based on preference.

Sequence Diagram 3.
fillGroups and addRemainingStudents methods

Filling groups with students is one of if not the most important process which our system does. 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.

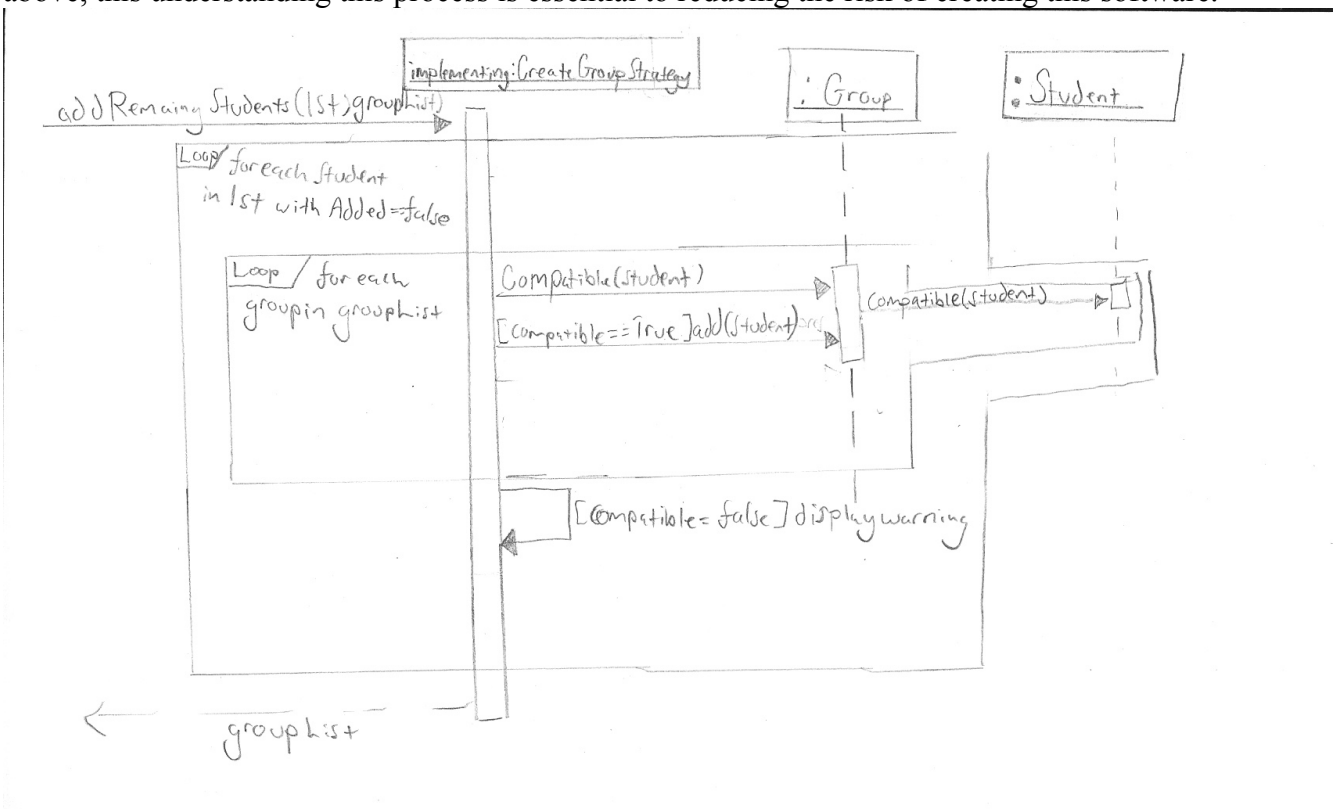


Here PDS stands for PreferenceDistributionStrategy. We create this sequence diagram as filling the groups is essential to our system, is difficult to understand, and we are not fully sure of how it should be done. This satisfies all three questions of the three Q's of architecture and therefore is extremely 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 separate parts of our system compartmentalized.

A preferenceManager class was created in order to manage what students the users did and did not want to be with/together. We created this class to promote cohesion and encapsulation, as it has the single job of handling preferences, rather than having for example Controller handle the preferences, in which case Controller would have multiple purposes and this would lead to bad software.

This sequence diagram above gives additional detail to what happens when students are to be added to the groups. This goes hand in hand with the above sequence diagram, and for the reasons mentioned above, this understanding this process is essential to reducing the risk of creating this software.



Therefore we believed that a logical layout of the interaction of this system was crucial. We strive for evolutionary design by using the 'strategy pattern' by creating an interface “compatible” for the groups to allow different ways to compare whether a list of students, or student is compatible with the group. This is evolutionary design as a user of this system which would prefer a different form of comparison can simply input a new compatible class which uses this interface. This can be used to implement compatibility based on skill and take into account the questionnaire which this system must account for. Our system will be better equipped to handle change in the future, should it be decided that groups need to be split up in other ways. The compatible interface uses the information expert pattern, as the group holds the most information about every member of the group. The compatible interface should then delegate calls to individual student comparison to the student class to be in keeping with the information expert pattern. The 'ACreateGroupStrategy' class is an example of a class which implements the createGroupStrategy interface, which uses the strategy pattern. We believed that it was important to understand how a possible group creation strategy would work in order to understand how our system should work.

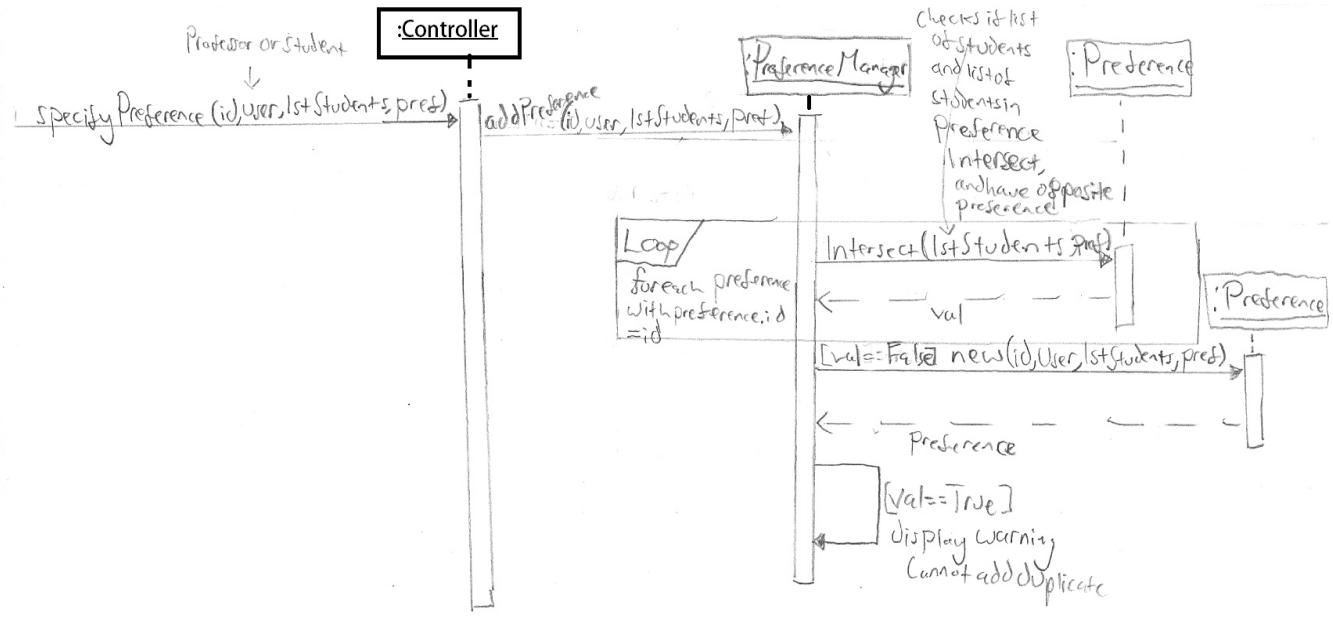
The addRemainingStudents method is passed lst, the list of all students in this course, and will be used

to add students to the groups which were not added when the GroupCreator added students based on Students' preferences and Professor's requirements. As shown in the diagram, the Group class has a method to determine a Student's compatibility with the rest of the group members.

The fillGroups method is passed the list of all groups, and a list of preferences, which it will then attempt to add to or separate into groups based on the preference.

We created the preference class in order to encapsulate the idea of a preference. Although, we later decided that this was not actually a good reason to have a class, as it only dealt with fields and had no unique methods, and we could instead use a collection to specify preferences.

Sequence Diagram 4.
Specify Preferences



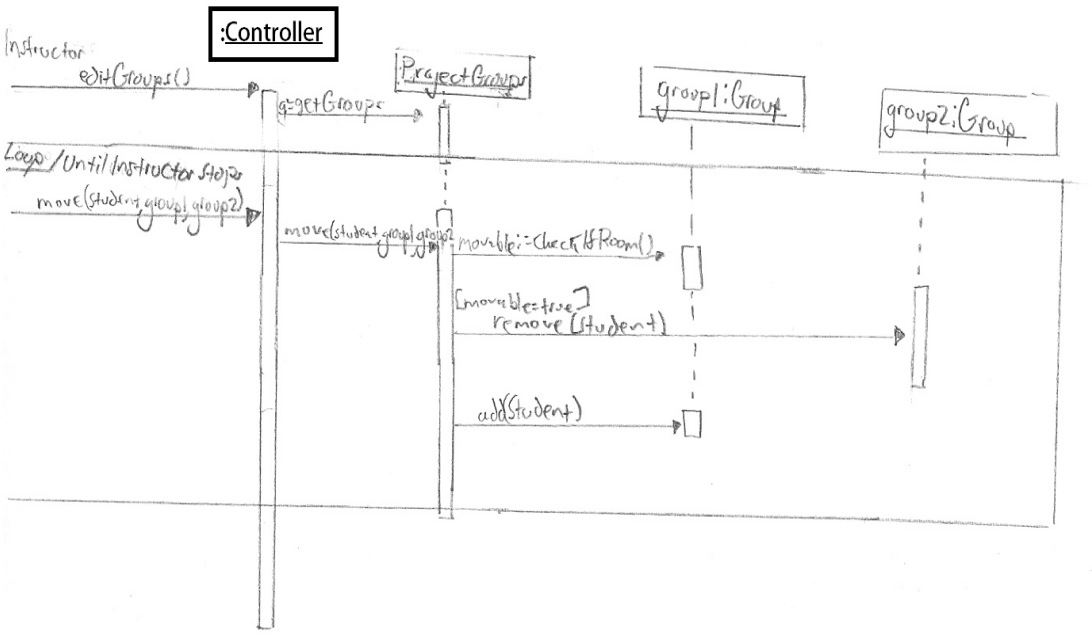
This sequence diagram shows the process of specifying desired (or undesired) groupings. An important feature of our system as seen in the feature diagram is the ability to specify preferences of groupings. It is also a piece of our architecture that a lot of our system relies on, and therefore to not 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.

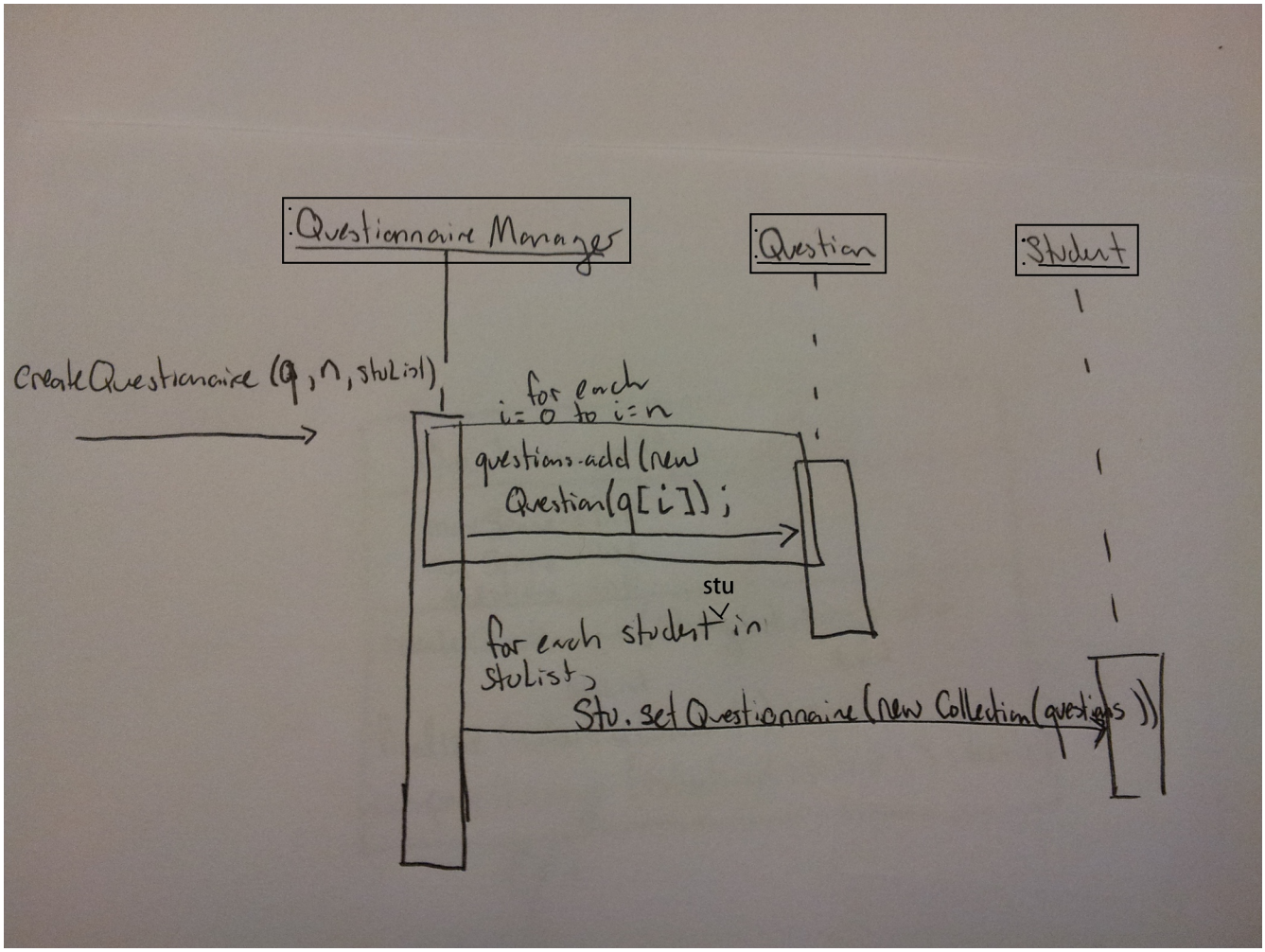
As stated previously we created a preference manager to ensure that each class has exactly one responsibility. The preference manager is the unique class which handles preferences. We believed that rather than overwriting, or adding any conflicting preferences it was best to just reject the addition 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 5.
Edit Groups.



This sequence is relatively straight forward, but as it is part of our executable architecture, we believe that it is essential, and therefore should be designed. We strive to decouple all tasks as much as possible by using information expert pattern, sending calls to projectGroups which holds the groups of all students for this project.

Sequence Diagram 6.
Specify Questionnaire.



The process of specifying a questionnaire is initiated by the createQuestionnaire method. This occurs as an alternate path of sequence diagram 1, in which the instructor specifies a list of questions (q) and the number of questions (n) and the system must create a questionnaire. This is not as essential as sequence diagrams 1-4, but we believed that since we group creation may rely on having a questionnaire answered, that this was indeed an important step to cover, and therefore would promote good architecture. We also believed that since we did not understand how the questionnaire would be

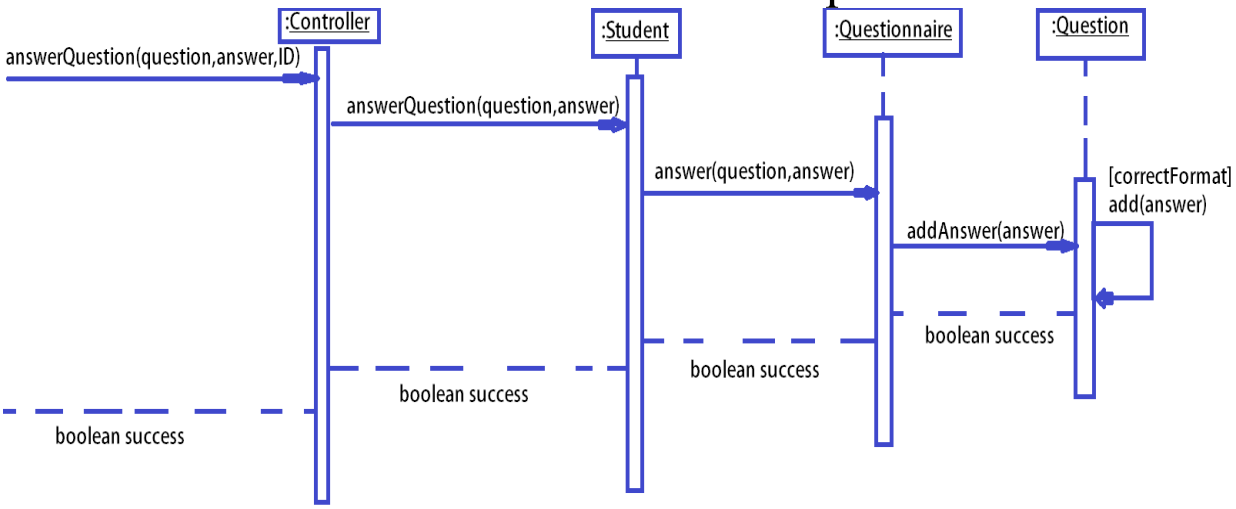
handled, that this creating this questionnaire sequence diagram would force us to do domain analysis on the questionnaire in general and allow us to have a comprehensive understanding of this part of our system, which we did not have previously. The QuestionnaireManager object maintains a list of all of the questions, and distributes them to each student as well.

We chose to have the student hold the questionnaire to follow the information expert pattern. The questionnaire answered by the student is the most related to the student and should therefore belong to the student. We created a questionManager class and Question class in order to ensure that each class has exactly one responsibility, and induce cohesion.

Sequence Diagram 7.
Student Inputs schedule Item

As this is the second most important part of the “student Specifies Preferences” use case, we believed it was important to do design for it. As well, we were not sure how to do it which satisfies one of the three Q's. We designed in accordance with the information expert pattern. We give the responsibility of adding the schedule item to the personal schedule as it has the most information about it. We do not focus on the case of retrieving the school schedule, as this will be done through the registrar system.

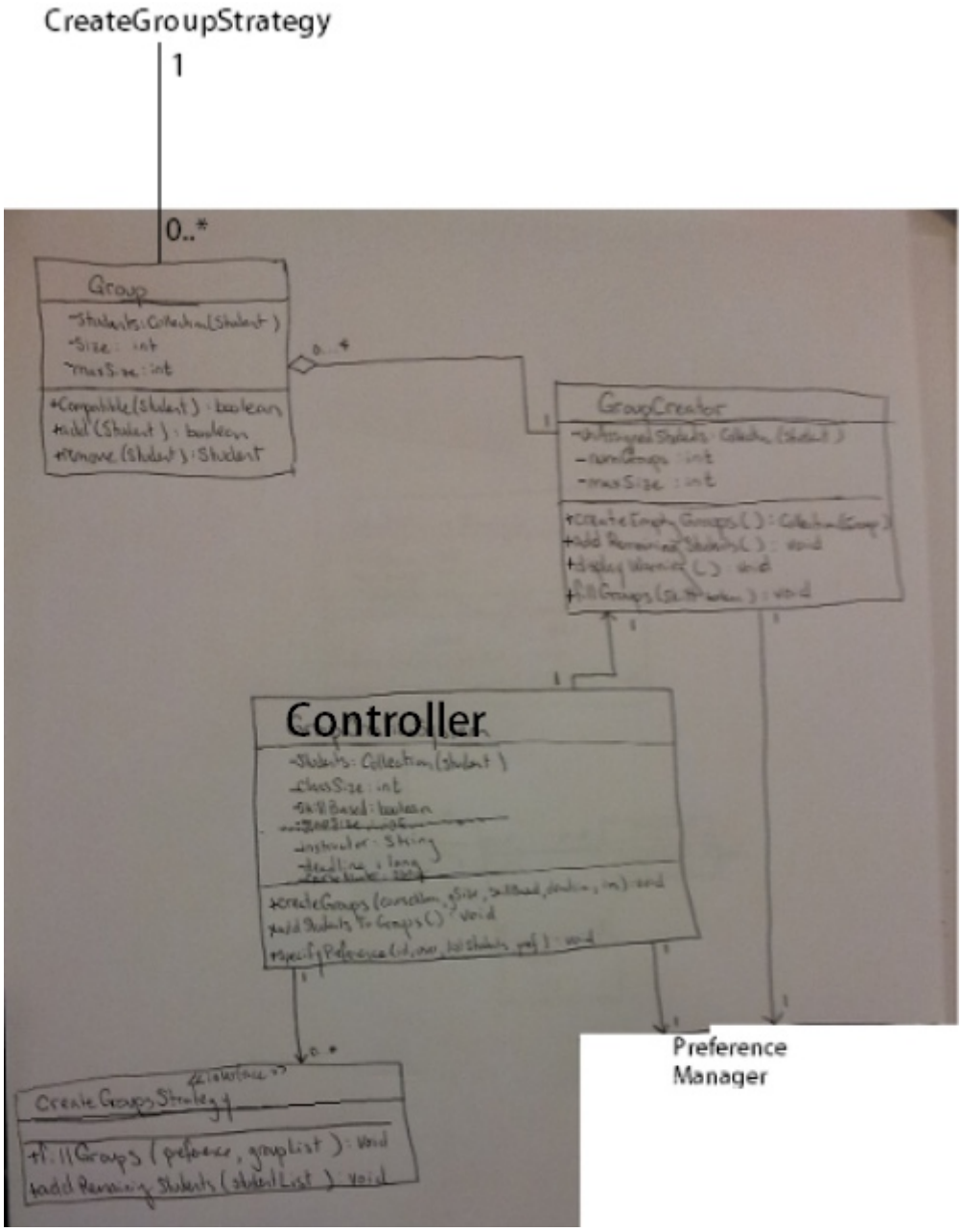
Sequence Diagram 8.
Student submits answer to questionnaire

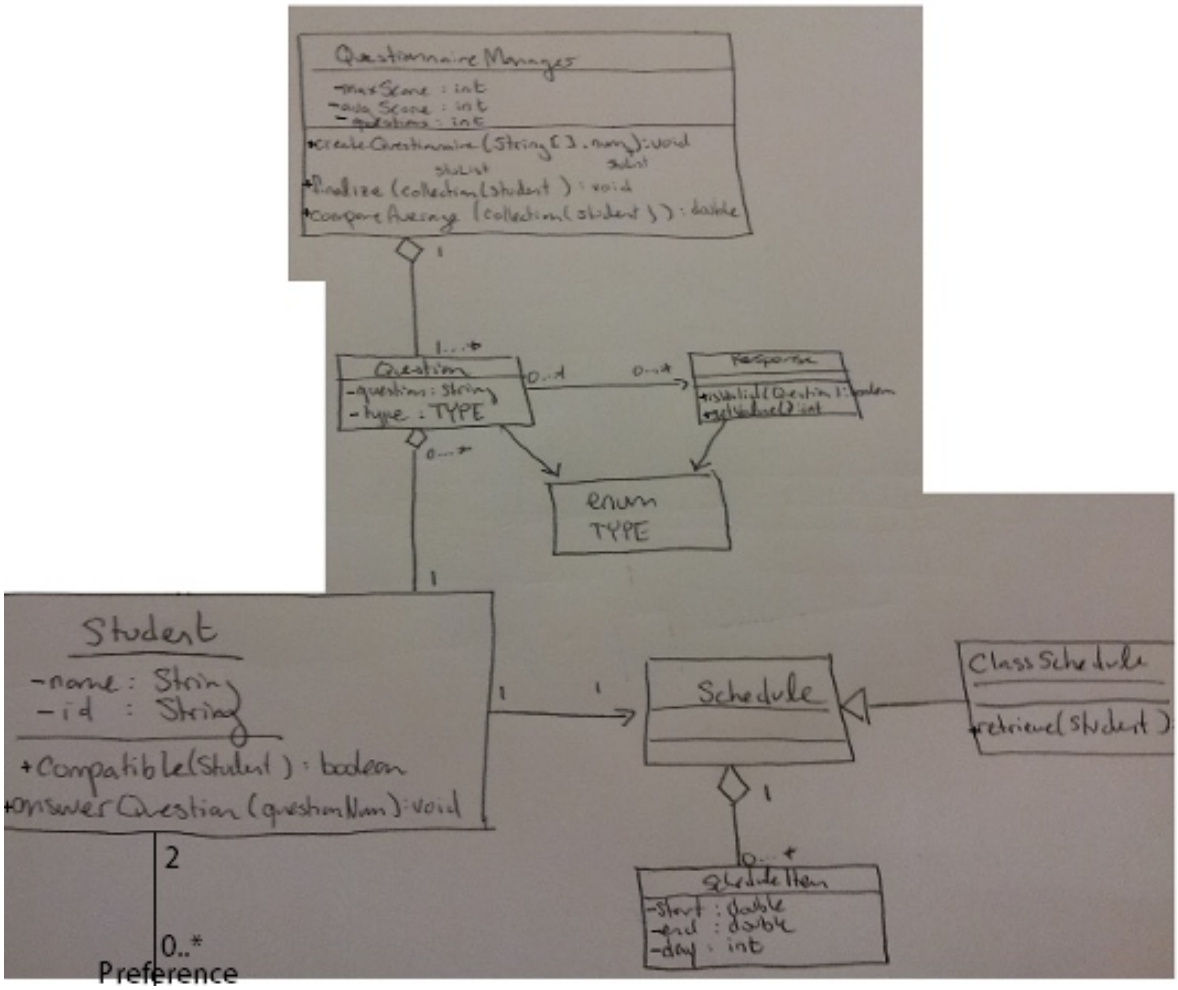


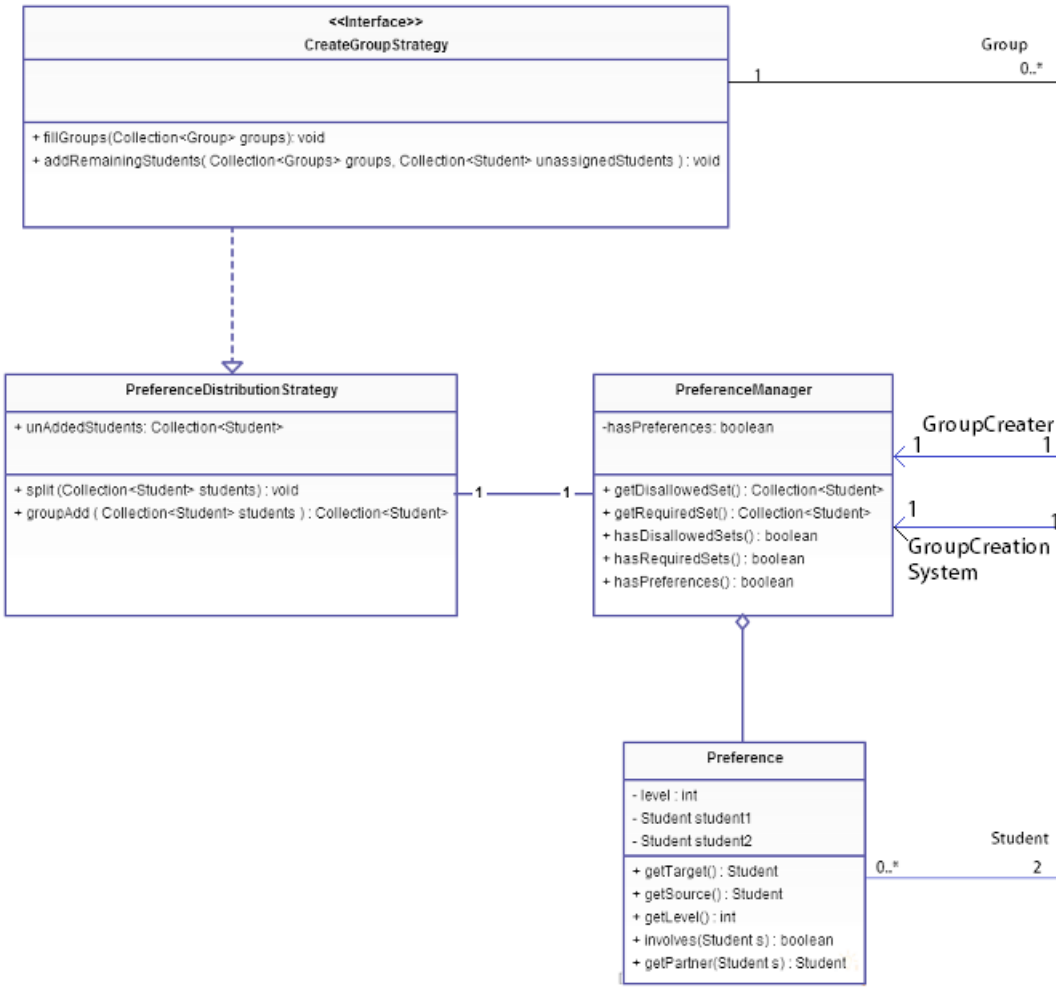
We design a sequence diagram for what we feel is the most important part of the use case student answering questionnaire, which is the the sequence of methods when a student submits an answer to a question. We do this design in order to gleam some structure of this should be done. Here we use the information expert pattern by making the question decide if the answer is in the correct format or not.

Class Diagrams

Due to the number of conceptual classes in the Domain model, we chose to represent the software classes using three class diagrams which we believe show the most important relationships, although disregarding a few obvious relationships, we believe that this will allow the diagrams to be more readable and make it easier to derive the essence of the class structures and classes from these diagrams. This allows us to see decoupling and possible modules in which to put these classes. This allows us to focus on a cohesive design in which each group of classes is mainly dependent on those in the same class diagram. For important relations to classes outside of these diagrams, we have included arrows leading off of the page labelled with the name of the class they are related to.


CreateGroupStrategy
1
0..*
Group
GroupCreator
Controller
Preference
Manager


QuestionnaireManager
-maxScore : int
-avgScore : int
-questions : int
+createQuestionnaire(String[], num):void
+finalize(collection(Student)) : void
+compareAverage(collection(Student)) : double
Question
-question : String
-type : TYPE
Response
enum
TYPE
Student
-name : String
-id : String
+Compatible(Student) : boolean
+answerQuestion(questionNum):void
Schedule
ClassSchedule
+retrieve(Student)
ScheduleItem
-start : double
-end : double
-day : int
1
0..*
2
Preference



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.

There will be a variety of classes which implement the CreateGroupsStrategy interface. The fillGroups method will add students to groups in a different way for each class which implements this interface. For example, one such class might fill the groups in a way which prioritizes skills over student preference, and another might totally disregard students' schedules. The GroupCreator method fillGroups will determine which object is responsible for filling the groups based on the inputted group parameters. For now, the fillGroups method simply fills the groups with students based on their position in the list and the maximum group size, only to ensure that the our code compiles and runs properly.

Also omitted from the Class Diagrams are getter/setter methods for each variable of each class, as well

as constructors. This was done to reduce clutter, but they will certainly all be present in the final code. Some of the non-trivial constructors have also been implemented in the code excerpts found at the end of this submission.

A thing to notice from the above diagram is the evolution from the sequence diagrams to the class diagram. After reviewing our sequence diagrams we noticed similarity between how the preference manager and preferences performed, and decided to encapsulate them into the same class, as essentially the preference class was just a container for fields, and therefore would work much better as a collection in the preference manager.

As well, after much consideration, we decided that the compatible method belonged better with the Student class. This method determines whether two students are compatible with one-another. This is due largely in part to the information expert pattern. Students have the most information about students. They have their schedules and the method is comparing whether students are compatible with each other, therefore we believe it was in keeping with the information expert pattern to make the student class responsible for the compatible method.

Change can be seen here from the sequence diagram in terms of what parameters are passed to what methods. This is due to utilizing the sequence diagrams as a tool to visualize how our system should work. After carefully reviewing them, we were able to pinpoint parts of our software which we believed was detrimental to providing reusable design and removed them, as well as correct encapsulation and responsibility handling.

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: Controller, 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:View, MovePanel, PreferencePanel,ShowGroupsPanel,StudentListPanel

Schedule: Managing student schedules is a big part of our system. As such, the Schedule module will contain all of the classes relating to the schedules.

Classes: Schedule, SchoolSchedule, ScheduleItem

Questionnaire: We thought it best to put the objects relating to the Questionnaire into their own module, since the Questionnaire feature was quite low on our priority list.

Classes: QuestionnaireManager, Question, Response