CS5030

Assignment 3 - Software Design, Modelling, and Analysis

190004947

November 21, 2021

University of
St Andrews

Word Count: 3,974

Contents

1 Functional and Non-functional Requirements

1.1 Functional requirements

- The system must be able to register users by school or by university by accessing the human resources unit and the student registry.
- Passwords must be set when a user first attempts to log in.
- Each school should have a 'manager' user who is a member of administrative staff and does not teach, supervise, or advise students.
- Every staff account should have relations to the school to which they belong, the modules they work on, and their office number (if exists).
- Every supervisor account should maintain a list of supervised students.
- Every advisor account should maintain a list of advised students.
- Every student account should maintain a list of modules they are studying.
- Student accounts can be marked as representatives and should maintain a list of students whom they are authorised to represent.
- Student representatives should be able to post feedback on behalf of other students that they represent.
- The system data should be updated once per week to account for real-world changes.
- The system should reset accounts each year while retaining data about feedback and response actions in a backlog.
- All types of user should be able to publish feedback.
- Users should be able to anonymise their feedback.
- Feedback is always anonymous when forwarded to relevant stakeholders by the administrator.
- Feedback should have a subject which can be a module, a level of study, or a user-defined subject.
- Feedback from students to staff should be forwarded to relevant staff using the feedback subject.
- Feedback from staff to students should be forwarded to relevant level coordinators to be disseminated to students.
- Every feedback should have a response time within which the relevant role holder should respond.

- Users should be able to edit or delete feedback that they have posted but only if a response hasn't been submitted yet.
- Feedback can be replied to with actions.
- Actions will state what is being done to address the feedback and will store a deadline for them to be completed.
- The author of feedback should be notified when a response is posted.
- The author of feedback should be notified on all progress on actions posted in response to their feedback.
- The system should periodically generate reports of all student-given feedback to be sent to school management.
- The system should periodically generate reports of all actions that have been taken in response to feedback to be sent to relevant students.
- The system administrator should be able to manage a universal configuration for the application which sets the default response time, report generation periods, and a toggle to enable reminder notifications of upcoming response deadlines.
- Feedback and response posts should be able to include both images and videos.
- Users should get a notification every time there is an update to a feedback they authored.
- Users should be able to choose their notification preferences.
- Users should be able to click a button on feedback submissions that states that they agree with the feedback.
- Users should be able to comment on feedback submissions. should be able to view unresolved feedback relevant to them sorted by soonest deadline.
- The system admin should be able to ban users for being 'bad actors', the definition of which is subject to the administrator's opinion.
- Banned users should still be able to view posts but should not be able to interact with them.
- Only logged in users should be able to view feedback, responses, and actions.
- Users should only be shown submissions relevant to them.
- It should be easy for users to see the feedback they have authored in one place.

1.2 Non-functional requirements

- Registration should simply ask for the user's university email address and then prompt to set a password which must be at least of length 6 and include letters, numbers, and at least one special character.
- Every get and post response should take less than 0.5 seconds to retain user engagement.
- Average page load time should be less than 1 second.
- No page should ever take 10 or more seconds to render as user attention will be lost.
- The content of a feedback post (including images and videos) should never exceed 200MB.
- Individual images and videos should not exceed 50MB.
- All implemented functionality must be compatible with all mainstream browsers.
- The application should be compatible with all versions of Windows, Linux, MacOS, Android, and iOS released after 2010.
- Critical system failures should raise an alert explaining the failure and then return the user to the previous state before the failure.
- The probability that a user encounters a critical error during a single session must be less than 1%.
- During updates or fixes, the system should never be down for longer than 12 hours.
- The database of user information should be encrypted to protect against sensitive data leaks.
- Users should only be allowed to post as themselves unless approved to post on behalf of someone else.
- Only the system administrator should be allowed to edit or delete feedback not written by them.
- The application shall be available in English as it is to be developed for an English-speaking school; should the system roll-out to non-English-speaking schools, support for new languages must be implemented.
- The date format across the entire system should be 'day month year' with the month being spelled out to avoid confusion.
- Users of all technical skill should be capable of posting feedback and finding relevant feedback within 5 minutes of using the application.
- Response deadlines to feedback should be obviously displayed on the feedback posts.
- Every action across the application should be quickly and easily reversible.

- The mobile application should share all functionality with the desktop application.
- All system components should be easily visible and accessible even on mobile-sized screens.

2 Ethical Considerations

2.1 Data security and privacy

In terms of data security, the only critical data that the application will store is user passwords. This is critical not only to protect the user's application accounts but to protect their other accounts as it is very common for people to reuse passwords across platforms. To protect passwords, they are to be encrypted before sent to be stored in the user registry database.

To maintain privacy, although feedback shouldn't contain sensitive data, the posts visible to users shall be restricted to that which is relevant. For example, posts regarding module CS5030 will only be shown to staff involved with that module and students currently enrolled in that module. The same goes for posts regarding specific schools.

2.2 Fairness of algorithms - avoiding bias

To avoid bias in the way that feedback and responses are displayed, we will implement a default view that will display posts by submission date; latest first. Alternatively, users should be able to sort feedback by popularity which is defined as the number of interactions from other users. Additionally, users will be able to sort feedback into resolved and unresolved categories. Feedback will be defined as resolved when the actions outlined in the response have been completed.

2.3 Consideration of unintended consequences

An example of unintended use is spam. If a user attempts to submit useless feedback multiple times in an attempt to litter the application's feed, the administrator will be alerted and will be able to put a post ban on that user's account. Submitting an excessive amount of feedback also puts strain on the staff who will then have deadlines by which they need to submit responses.

The administrator may decide to put a post ban on a user's account for other reasons such as making a rude comment or generally insulting the way in which a module or school is run.

2.4 Fairness of working practices

To remain fair in the working practices of the platform, we aim to create an inclusive environment where people feel heard and respected. By having response deadlines for all feedback submissions, students can post confidently knowing that their concerns will not be ignored or overlooked.

3 UML Use Case Diagram

Figure 1: UML Use Case Diagram

4 Use Case Specification

4.1 Use case name

RespondToFeedback

4.2 Brief description

Staff responds to feedback submitted by either a student or staff.

4.3 Actors

Staff

4.4 Precondition

Feedback has been submitted.

4.5 Main flow

- The use case starts when the staff member selects a posted feedback to reply to.
- They can then enter text like commenting on a social media post to reply to the feedback giving relevant information. Images and videos can be uploaded here too to supplement the response.
- Once the response has been created, the staff member will click a 'Post' button which will submit the contents of the response to the application server.

4.6 Postcondition

The feedback has been responded to and the feedback author will be notified.

4.7 Alternative flow

The alternative flow follows the same logic as the main flow with an extra step between step 2 and step 3:

- If there are actions to be taken to resolve the issue noted in the feedback, the staff member will click an 'Add Actions' button which will reveal a new text box. In this text box, the staff member will detail the actions that will be taken and choose a deadline.

5 UML Class Diagram

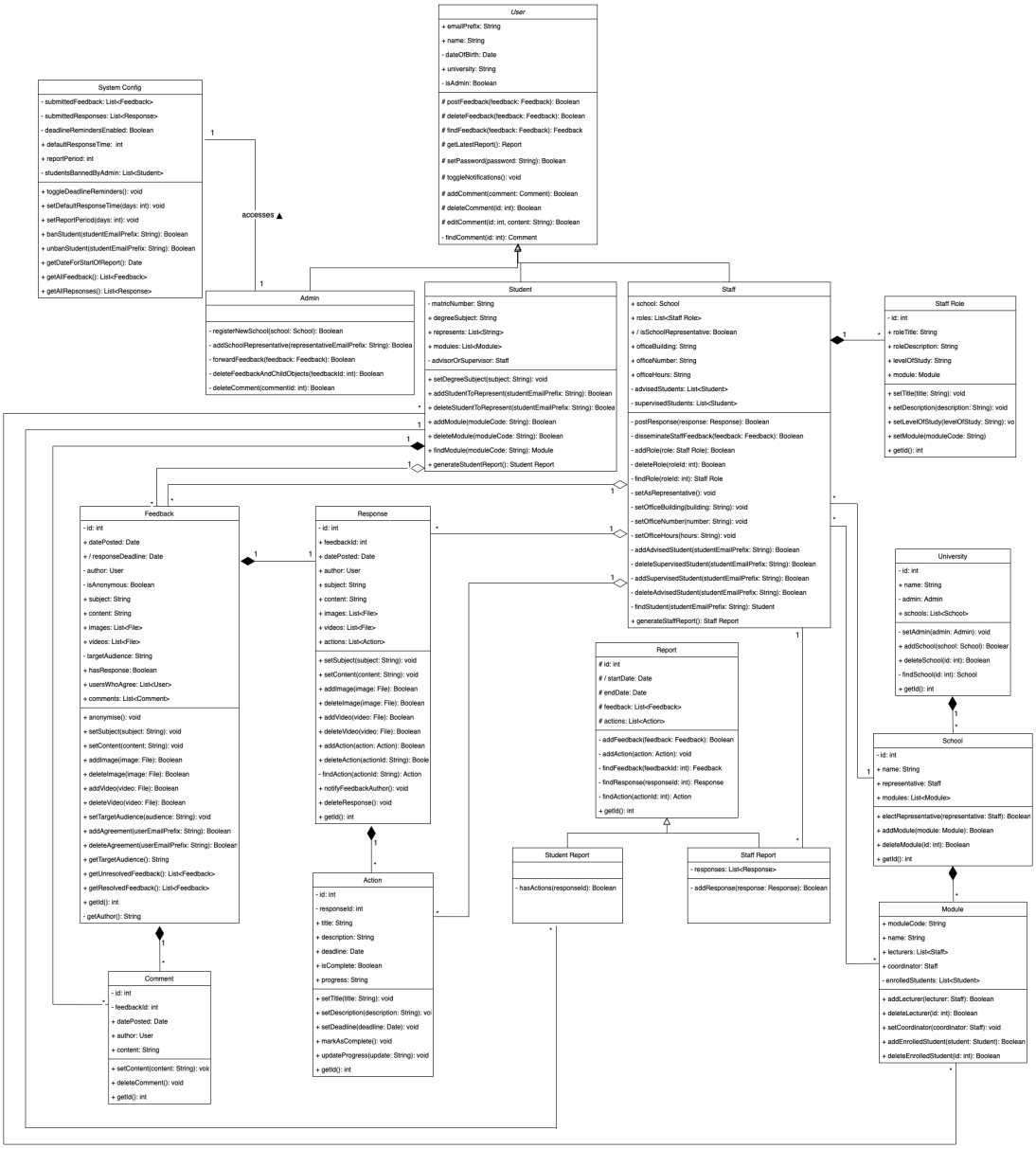

Figure 2: UML Class Diagram

6 Behaviour Design for an Interaction Sequence

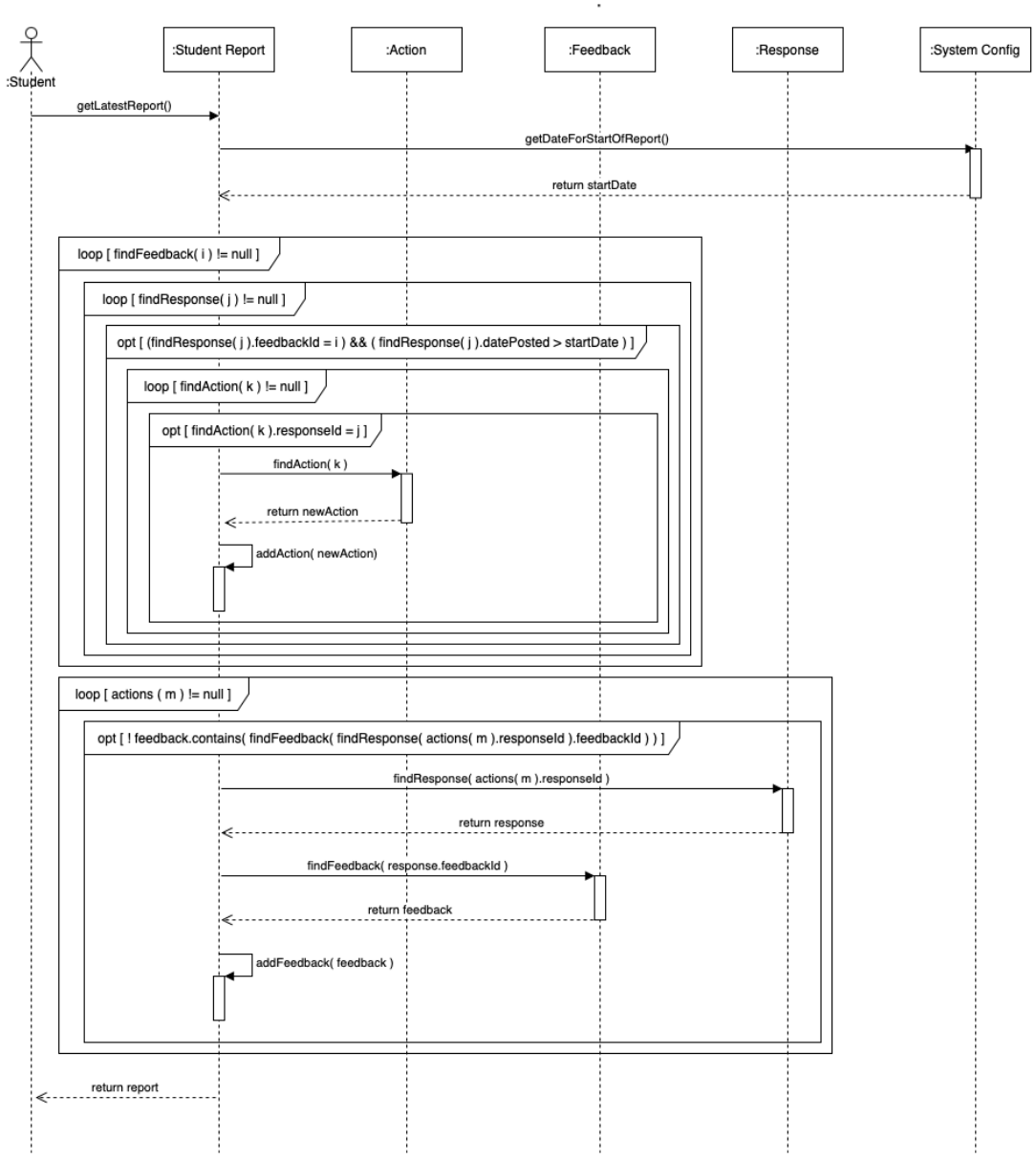

Figure 3: UML Interaction Sequence Diagram

7 Design Analysis

The design I created for the feedback and response application is outlined in the UML class diagram above. The architecture consists of sixteen classes which represent five main logical systems; user accounts, administration, feedback and responses, reporting, and a university structure.

I have set out three types of user, admin, student, and staff, all of which are subclasses of the user class. The administrator type is identified by an 'isAdmin' flag in the superclass and does not have any additional attributes. This design assumes that the administrator account's sole purpose is to maintain the application and not to interact with other members on the platform. The rationale behind this is that system management and configuration can be safely restricted to one person. The admin is not equivalent to the school representatives who are members of staff with staff accounts. One role of the admin is to register new schools of staff and students, setting a representative for new schools, so that the application can be incrementally introduced to a university. As required by the specification, the administrator oversees the system configuration which includes attributes such as the default amount of time staff will have to respond to feedback and the period of time that automatically generated reports will cover. Extra responsibilities of the admin that are beyond the specification are the ability to delete feedback (and thus all child objects) posted by staff or students and the ability to ban students. These two functions allow the admin to maintain a clean platform by removing bad actors or malicious submissions.

When a new school is registered, all staff from that school will be signed up using data from the human resources unit and all students will be registered using data from the central registry. Staff and students are specialisations of user objects with extended attributes and operations to allow fair use of the platform. For example, students can set others as representatives who will then be able to submit feedback on behalf of them. This is achieved by implementing a list of other students as a student attribute called 'represents' which can be modified using class operators. Instead of assigning users unique IDs, in my design users are identified by their institution email prefix, e.g. jr263. Where IDs are required in other classes in the architecture, there is an attribute called 'id' which stores an int type. The idea is that each subsequent instance of each class will take the ID that is one greater than that of the instance before it. In practice, this wouldn't be an efficient way to keep track of objects as there would have to be global variables somewhere keeping track of active instances. However, for this logical specification, it allows a simple mechanism for generating periodic reports for both students and staff.

The reports produced for staff and students are different and hence I have defined 'report' as a generalisation of both. The student report will show all the actions that have been posted along with the feedback they are in response to. This is in line with the specification's requirement to produce "you said, we did" reports for students. Since actions are their own class with 'progress' as an attribute, this report will fill up with everything that staff have implemented as a direct response to feedback on the platform. The staff report needs to be more comprehensive

for school management to be aware of progress as well as new issues that may have arisen. This report is generated simply by collecting all of the feedback that has been posted over the most recent report period (as set in the system config) as well as any responses that have been submitted. Using the ID mechanism described above, this information can be collected simply by iterating through IDs and checking if the post date is after the start date calculated by the report. The weakness here is that if and when a feedback object is deleted, it's ID gets freed up and must be reallocated or otherwise dealt with before a report tries to generate. As visible in the sequence diagram above, report generation will iterate through feedback and response objects until an ID returns null. Since responses are tied to feedback through their ID, we can't simply bump all object IDs down to compensate. One solution to this problem would be to allow access to the getAllFeedback() method in the system config class which will return a list of all existing instances which can be iterated over. The freed ID numbers can even be reassigned by adding deleted IDs to a queue structure. When a new feedback is posted, if the queue isn't empty, then the ID will come from there rather than the next integer above the last posted feedback ID. In a realised implementation, the report generation algorithm could simply loop through all existing feedback objects currently stored in the database rendering ID iteration unnecessary.

In alignment with the specification, posting feedback is a method in the user class as to be accessible by all accounts in the application while posting a response is an operation in the specialised staff object to restrict permissions. When a staff member posts a response, they must include a subject much like in the feedback objects. This subject will identify the relevant stakeholders of the post which can be interpreted by the administrator for passing feedback on. Much like the admin forwards these posts, level representatives are able to push staff feedback to students which uses the findStudent method with relevant email prefixes from the school and module objects as a parameter. The system design defines feedback objects as a sort of container which will store its own data as well as a link to a response and comments. While actions are also part of the response object, upon deletion of a feedback object, the operation is propagated to the response, the comments, and the actions. This is correct behaviour and is a useful structural constraint that removes the need to search for children objects.

Actions exist as a composition with responses (which in turn are compositions with feedback) as actions cannot exist outside of a response (much like responses cannot exist without a feedback to respond to). Actions are tethered to responses and not feedback since the hierarchy connects the two objects anyway. A single response is able to have multiple actions associated with it should that be necessary. Each of these actions stores a title and description as well as a deadline date, a flag for completion, and a string attribute which will keep track of progress. This progress string will make up the large majority of student reports. Since actions are posted with responses rather than after them, it suffices to have a notifyFeedbackAuthor operation in the response class which will send a notification to let the original author know that there has been an update to their submission.

I have extended the specification in the feedback and response system to enable additional user engagement. Users can mark feedback with an agreement; functionally this is identical to 'likes' in a social media application. This is facilitated simply by a list of user objects which

is accessed by addAgreement and deleteAgreement methods. As well as this, students are able to post primitive comments on feedback. These comments exist as their own object with a many-to-one relationship with feedback and will store the date posted, the author, and the content of the comment. I included this functionality as part of the system design to extend the student voice from individuals and representatives to group discussions. Students collaborating in this way can build up a more comprehensive idea of whatever issue the feedback is addressing. Furthermore, both the feedback and response objects have been expanded to enable image and video upload to supplement the notions being portrayed.

In modelling the structure of a university, I have three classes to represent the hierarchy of an educational institution like St Andrews. A university object is composed of school objects which is in turn composed of module objects. This design means that both new schools and new universities can be introduced to the application. I have defined staff to belong to singular schools while students are not bound in this way. This is because students can take modules from different schools but staff members will be specialised to one. On the other hand, both staff and students have many-to-many relationships with module objects. There exists a duplication of data here for ease of access; while module objects store a singular coordinator and a list of lecturers, each staff member will also have relations to one or more 'Staff Role' objects which store an extension of this information.

Staff roles include a title, a description, a level of study, and a module, the last of which can remain empty if defining the role of a level coordinator. This structure enables flexibility of the system to account for complex staff roles and even shared roles. Other than the id, staff role data has visibility set to public as everyone in the university should be able to lookup the relevant responsibility holder to address their concerns.

Of the three user types, both admin and staff keep the visibility of all their methods private as they are the only ones who should be able to manage their account. Contrasting this, student users have public visibility for all of their methods. This is because the specialised student operations are for maintaining their enrolment status and details such as which modules they are taking. This information is likely to be imported from the registry and is not for the students to manually enter themselves. The visibility of other class methods is determined by which external objects need to access them. For example, all system config methods are public but will of course assert that the user calling them has the isAdmin flag set to true before running any system critical functions.

8 A Brief Reflection on UML Diagrams

8.1 Merits

The widespread use of UML diagrams is due to its powerful expression of both simple and complex system design. In industry, one of the main purposes of UML diagrams is to communicate the architecture to less technical stakeholders. For example, through a class diagram, the system designers are able to communicate to perhaps a product owner the functionalities that will be implemented and the relationships between different logical component systems. In the case of the feedback and response system, although the class diagram may look dense, upon inspection a reader is able to determine the different types of users and their scopes as well as information such as what types of data will be included in both feedback and submission posts.

Another benefit of using UML diagrams to document the design of the system is that it maps one-to-one with an object-oriented implementation. Using an object-oriented language like Java, classes can be created for each box seen in a class diagram and an interface can be created extremely quickly simply by including the elements from the boxes in the diagram. In some cases, a large amount of boilerplate code could be generated directly from the UML depiction. Since the class diagram includes both data types and visibility scopes, a lot of implementation logic is implied which is extremely helpful for the programmers. In the case of our system, the hierarchy of feedback and response objects describes exactly the dependencies to be implemented.

By organising objects and their relationships in this way, system designers can identify structural issues before implementation even starts. The class diagram seen above is a result of multiple iterations each of which improve on the previous version by fixing a logical inconsistency or by adding a workaround to run required functionality. Without initially producing these diagrams before development begins, hundreds of hours could be wasted writing code that will eventually be either scrapped or overwritten. Again linking this to the context of our system, most of the functional requirements could be implemented and working before the developers considered how the application would be rolled out to additional school and additional universities. Once this issue is encountered during development, the structure of individual objects might not accommodate the introduction of such new systems which can lead to extreme unforeseen costs. To contrast, when building the UML diagram, issues of this nature would become apparent and can be solved by simply changing relationships, editing attributes, and adding methods to objects.

Another extremely useful result of building out an architecture in UML before implementation starts is that roles can be explicitly established within a development team. For instance, a team of five developers could each tackle one of the logical systems of the feedback and response application as defined in the design analysis section. Once a developer completes the implementation of their subsystem, as long as their outward facing interface follows the specification, they can easily transition to help other programmers implement different functionality. The class diagram also gives the product owner and product manager an idea of the project scope. It can

be extremely useful in estimating both development time and cost. To avoid going over-budget or missing a deadline, the system architecture could perhaps be simplified to meet budget or scheduling constraints.

8.2 Limitations

Although UML diagrams introduce all these benefits to the development cycle, there are also limitations. For example, it is very common for requirements to change over time which could cause conflicts in existing implementation since it has been matching a now outdated model of the final system. In the event that this does occur, many hours of useless development could have gone by implementing functionality that is no longer required when there may be new requirements that haven't even begun to be addressed. For a small system such as our feedback and response application, this is less of a problem and can be avoided if the project manager actively manages and priorities requirements.

Another limitation of using UML diagrams to document the system is that it is not very compatible with an agile development paradigm. In agile development, the system requirements are constantly changing to accurately represent to most important features that the system should include. When planning the architecture in UML diagrams, it is important not to over-design the system and commit to a blueprint that is beyond the scope of the development team's sprint cycles. Furthermore, the team could eventually decide that there is a much simpler and more sensical way of implementing a certain subsystem but due to the already implemented interfaces this could be highly challenging to adapt to. On the other hand, UML is not completely useless in agile development and basic diagrams portraying overarching principles can be agreed upon during scrum meetings and subsystems can be quickly implemented during sprints by following a class diagram.

In terms of interaction modelling, sequence diagrams like the one above can be valuable in expressing the flow of system components to achieve common goals such as generating a periodic report for students. However, modern computer interactions can quickly become highly complex with many series of actions all achieving one goal. On top of all the potential errors that may occur, a comprehensive sequence diagram can become convoluted and illegible to non-technical stakeholders. It might even be easier to implement entire sets of functionality without trying to follow the interactions set out in a sequence diagram. In my sequence diagram above, I have taken a high-level approach that closely imitates how the implementation will run the process of generating a student report. This is useful for the development team but doesn't include alternate routes for reporting and dealing with all the different types of errors that could come up. If the error handling is not correctly considered, user experience of the system can be completely destroyed and the application engagement will drop considerably regardless of the amount of work that has gone into implementing functionality.