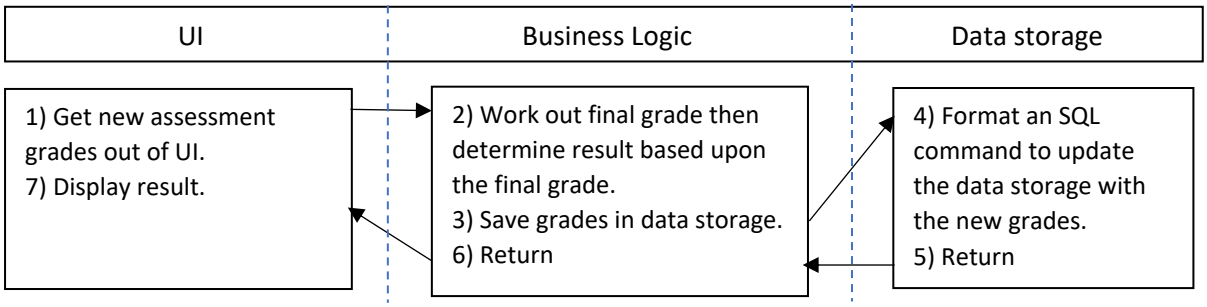
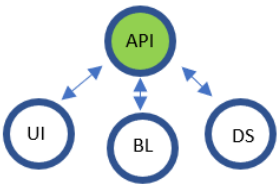
3 Layer Approach

To keep each of the layers interchangeable, the original implementation of the 3-layer approach used in the project was designed as follows, with calls all being routed through the business layer.

Scenario: Changing a student's grade and determining new result for a single module



The issue with this design was that trivial methods were needed to proxy calls from the UI to the Data storage in the Business logic e.g Student ID needed for UI display, [UI->BL->DS] ->[DS->BL->UI]

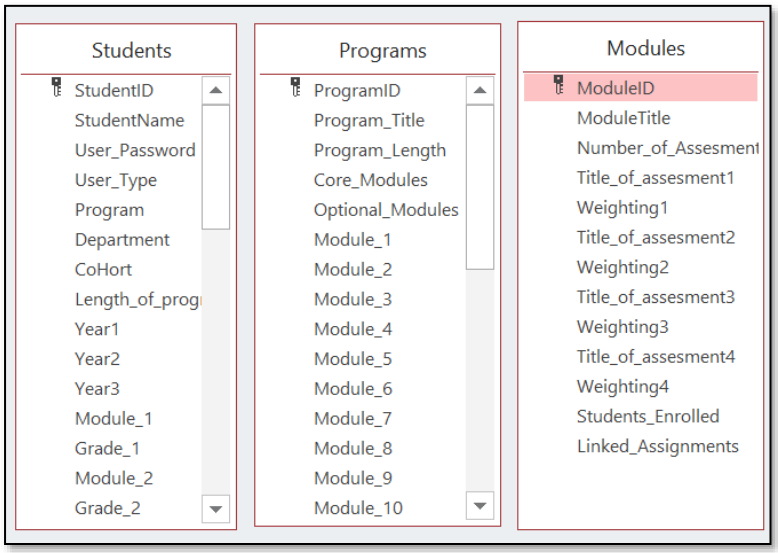


The design adopted later on in development [figure left] makes use of a centralized node (API) to control the flow of data, not only reducing the total code created by 200+ lines but also furthering the ability to interchange components with only a change at the API level.

With this layered + API approach it would be a miniscule amount of work to adapt the code to a new data storage medium without affecting the functionality of any of the other layers as long as the new instance implements the layers interface.

As a general rule each layer can be categorized by the following, if:

1. It makes use of UI components and is based solely around Input and output of data.
    a. UI
2. It revolves around processing data into another form/ performs system critical methods
    a. Business Logic
3. It must make immediate access of the data storage medium to read/write.
    a. Data storage layer



[Figure above] Primary keys and attributes of each table.

Data Storage (Datastorage_class.cs)

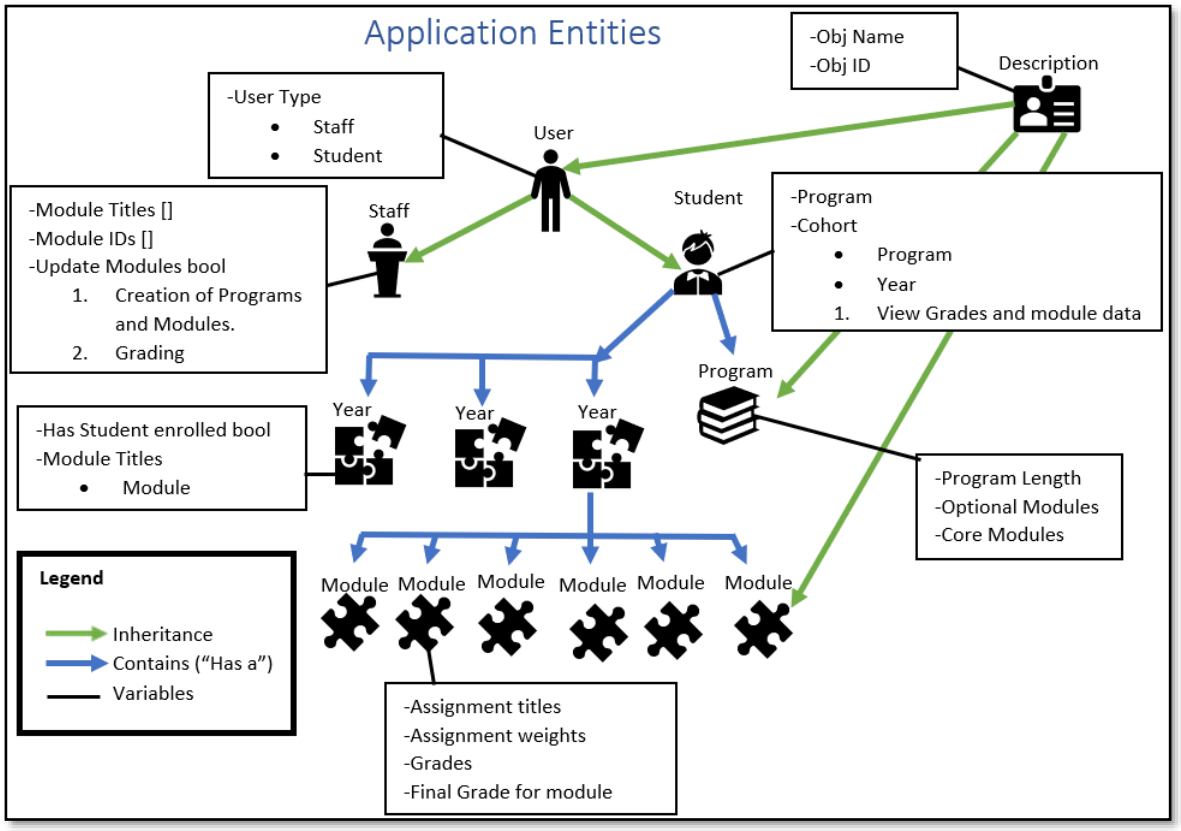
The data storage layer consists of 3 Tables (Students ,Modules, Programs). Each row describes an object of the tables type (a row in the students table describes 1 student). The approach used to develop this layer was focused on relating data and building relationships at a class level rather than at a data storage level to keep said layer easily interchangeable. To that end the database was kept simplistic.

User Interaction design

Each tab/form is consistent and intuitive in design, each page has data entry fields (textbox, checkbox, …) located on the upper segment of the screen with action control buttons on the bottom half (clear, commit, preview). The UI has been intensively tested and has many error recovery catches and checks to minimize the system crashes and it should be considered reliable.

Data design and translation to business logic objects:

When data is needed a connection is opened, data is read, the connection is then closed and finally the data is sent to the invoking method that requires the data. This minimizes the length of time the database is kept open (connection also pooled). Data is either added to a complex structure or manipulated and discarded after use. The complex structure detailed in the figure bellow was designed to minimize calls to the data storage layer, when working with a student's data in the "Students tab" to improve efficiency when jumping between various module values.



State Invariants / Identifying attributes of each at a class level include:

1. **Type difference**, as they're all class objects and not primitives (a module is of type module, year of year, student of user, staff of user, program of program)
    a. A module is always a module if it is of type module.
2. Each must have a unique ID.
3. Students and staff members must have a "user type" value (enum) indicating its type which can be used to distinguish them.
4. **Obj ID length**, each object inheriting from the description class has a different sized ID length (User = 10/7 (student/staff), Program = 6, Module = 5)

At a Database level:

1. Different types of users are identified by the 4th column in the "Students" table, this holds either "Student" or "Staff"
2. Modules and programs each have their own tables respectively.

Business Logic (Business_Logic.cs)

The Business layer contains all system critical functions. The general structure of the layer is based around the idea that “UI calls function X to perform a task ,upon which function X calls X1,X2,X3… until it has completed the task” by making use of methods within both the BL and DS layers.

Due to this layout of methods it is relatively simple to replace method X1 or X2 but quite complicated to reimplement the entire X function that is called by the UI.

Included in the business layer are 7 class objects that form the Module, Program, Student, etc entities. These entities are stored as instances on the UI layer but are created and manipulated on the business layer. Each contain methods to retrieve and set their values along with validation on creation.

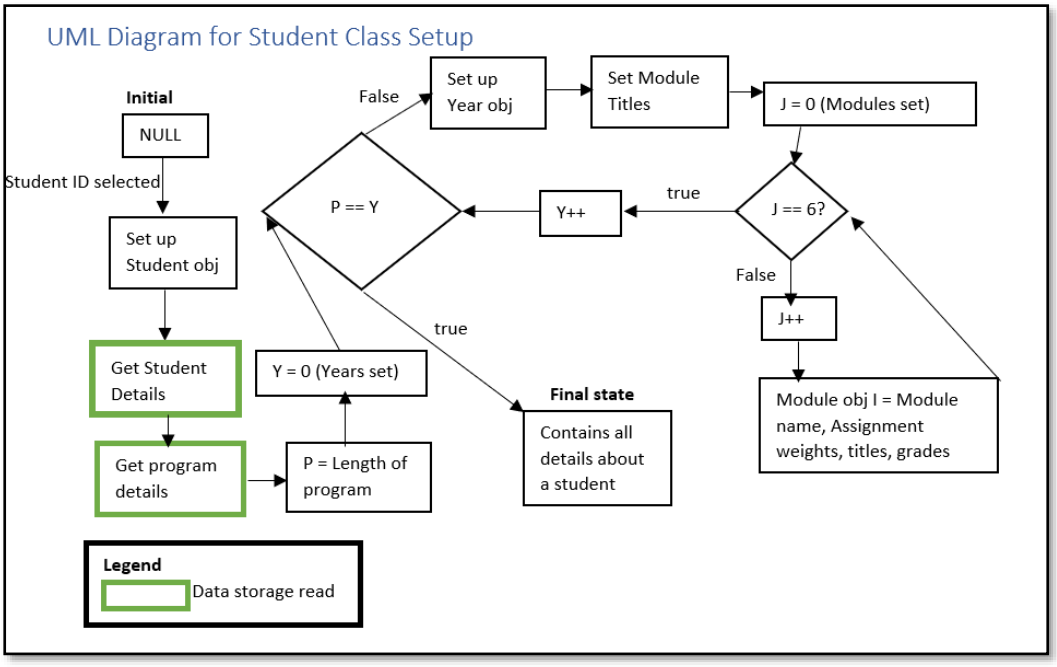
The business logic class implements the “IBusiness_Interface” which outlines all callable system functions with comments on how to use them.

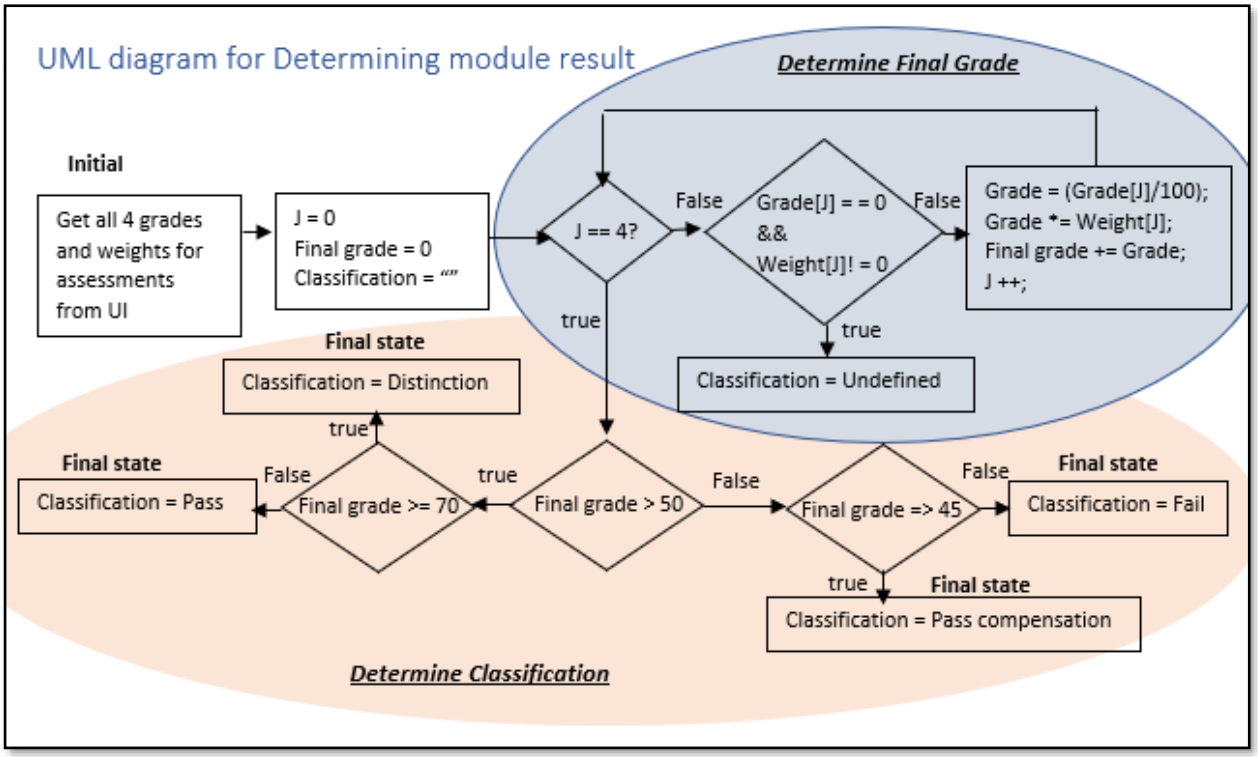
Similarly, with the data storage class implementing “IDataStorage_Interface”.

Validation

Checks for the validation of data are conducted on all objects in the application entities figure ,all objects have business layer validation checks , some also have UI layer checks along with Business layer checks.

Requirement	Implementation
Modules, Users and Programs have IDs. Ensure they conform to outlined rules.	Class “Description” is inherited by all objects with IDs used in the system, it has a virtual method which checks if the length of the ID is equal to the required length specified in the given parameter. Overridden version used in Student class to ensure Cohort rule applies.
No two programs may have the same set modules	Similarity method implemented in business logic that compares all core modules of one program to another and returns a similarity metric (% similar). If one program is > 80% similar it is not created.
Some modules are determined, and some are optional.	The User is enrolled onto all core modules of a program at registration. When making use of the system the user must first continue to enrol onto his selection of the optional modules before he can view assessment information.





Reflection

To the best of my understanding, all functionality and requirements have been implemented and I believe the client would be satisfied with the implementation.

In terms of being able to do ”task X” I believe my system accomplishes each task well. In terms of good design, I feel the vast majority of the code is modular with specific emphasis on the data storage layer. The use of the API furthers this ease of replacement ideology. If given the opportunity to start again I would focus on the manipulation of the User type objects, over 90% of the code to manipulate these objects is within the BL layer however there is some reliance on code within the UI layer (as this is where their instances live) that would make a re-implementation of the UI layer with the current BL layer more complex than it needs to be. Moving the instance form the UI to variable within the instance of the BL would be far more appropriate in terms of design.

Update: (fixed, moved instances, they are now stored on Business Logic layer)

Known Issues

A memory leak is present in the creation and closing of forms that I have been unable to fix, despite disposing and closing of specific forms their remnants remain in memory. Furthering to this known issue, threads for these forms are still active after program termination.