Mark Tracker

# Business Context

Marking assessment tasks are a chore, especially when it comes to book keeping information. There are 3 primary pieces of information that are essential to the marking process:

* Students - those who submitted
* Marks - the raw marks obtained by each student
* Feedback - suggestions for how the student could improve, or where they lost marks

Complicated assignments make this information hard to retain in a systematic way, which as a result makes it difficult to book keep and relay the results back to the students in an efficient manner.

"Mark Tracker" is a GUI desktop application that shows the **structure of an assessment tasks**, in terms of its components. These components are defined by the assessment task type - for example, if the assessment is a quiz, then a component could be a question, or a sub-question.

With each assessment task, and each associated components, the application is also able to show **marks achieved by students**. For the assessment task, the application can display some statistics related to this matter: average mark, best done component, and mark distribution.

The application also allows users to preserve **feedback**. This feedback can be related to the assessment, components, or the student for that assessment. For the latter, the application allows users to preserve feedback for each student for EACH component in the assessment task.

This application is made to ease the feedback distribution process, so that the markers don't need to scramble through an assessment or paperwork to determine specific feedback or marks for a student. By preserving this information on the fly, the application gives a systematic way of achieving this task. The end users are ultimately teachers who are marking assessments, who wish to preserve, retrieve, and examine information in a systematic way.

# Requirements

Core and optional? Future? VERY FAR into the future?

The application should run on the Windows operating system.

### Core requirements

* Creating data  
  Users are able to create assessments, components, groups, students, and SMI.
* Viewing data  
  For each assessment, users can view its details and the associated components.  
  For each component, users can view its details and the associated groups.  
  For each group, users can view the associated students.  
  For each student, users can view their associated SMI's.  
  Users can view an individual student's SMI for particular assessments and components.
* Users can change the view of the assignments. For example, sorting them in order of date due, or weighting. They can also select what entities to view regarding the assessment task. For example, they can "toggle" to view components, groups, and students. This way, they are able to customise the depth of information they'd like to see. (See GUI for more info).
* Modifying data  
  Users can modify details (through text editing fields):
  + Assignments: Due date
  + Component: Mark available
  + Group
  + Student
  + SMI
* Removing data  
  Users can remove assessments, components, groups, or students.

### Additional requirements

* Users can view a graphical distribution and statistics of marks for a particular assessment, with regard to a component, group, or students.
* Users are able to attach assessment task specification documents (eg uploading PDF)
* Users can create a list of students by importing all their IDs
* Users can export assessment results in certain formats (eg csv)
* Users can choose a database file to load from, as well as changing the default database

### Future

* Make the application into a web system, with two parts: a central repository of assessments, components etc that contains all the information, and a client-viewing side (the application developed will be the prototype for this part).
  + This will also require a 'log-in' mechanism, so that each teacher has access to their own classes etc.
  + This also supports "sharing" of information (ie permissions) of which assignments are viewable / changeable by certain teachers.

### VERY FAR into the future

* Develop a compatible "collection" system, where students can collect their given marks. The end users will be the students themselves.

# GUI Design

Screenshots, flow between each screen

# System Design

High level system design (eg database or not?). UML Design: classes for each terminology, Technologies that will be used for implementation, Rules

### Design Diagram

[ The one with the UI, application, and data layer ]

### Application File Format - storage and retrieval

The application will use **SQLite** - a lightweight application that is great for storing desktop application data. The application will need to specify its own **application file format** that contains ALL the entities, and can consult things on the fly.

What is currently unknown:

* Do the end users need to install SQLite in order to use the application?

Some research has been done into the *flat file database*, either specifying my own format, or through a standard hierarchy format eg XML. But these are terrible ideas.

* Either the entire file(s) need to be read into memory, or the files are consulted on the fly. The former is not scalable for data, and the latter has reduced performance implications.
* There will be MANY writes and updates. This will involve seeking through the file, which has terrible performance implications

### Technologies used to implement

* GUI and coding: Using Microsoft Visual Studio, coding in the language of C#
* Database: SQLite

### Rules and Assumptions

### Function rules

* Creating an assessment:
* Data in the PGP should be populated based when a course is selected from the AP.
* VEP displays information based on the following cases when they are double-clicked:
  + Course: Details about the course
  + Assessment: Details about the assessment
  + Component: Details about the component
  + Group: Details about the group
  + Student: Details about the student + their mark for the selected component
* Deleting a student: Student will be associated with one assessment. Thus removing the student will remove their presence in all associated components,
* Exit button: Will prompt to "Save" if there is unsaved modifications

### Database Design

Entities:

* Course:
* Assessment:
* Component:
* Group:
* Student:
* SMI:

Entity Relationship Diagram:

### Solving Impedance Mismatch

This exists because SQLite is a relational database, and C# is an OO language. The decision was to go with an **object relational mapper** (which can be achieved using Entity Framework 4 in C#). There are several reasons:

* Better in long-run for the application development process. Hardcoded SQL statements are quite annoying to maintain and implement.
* Application isn't performance critical (although ORM's don't usually affect performance significantly)

The alternative decision is to have an **application layer** that abstracts the retrieval and conversion of data to and from the database. Essentially, drawing the data and conversion between types are very application specific. In the context of this program, extracting from the data layer needs to come in several flavours: application structs, APNodes, and PPNodes. As a result, there are several options:

* **Option 1: Single data layer class**  
  Data retrieval and conversions all occur in a single class. This gives very high abstraction, single point of DB entry, and also easier testing. However, it is not too modular (very application specific), and there is redundant code when it comes to retrieving essentially the same records but just parsing into different classes.
* **Option 2: One conversion class for each type**  
  Each conversion is defined in its own file. This is quite modular, and code can be inherited. However, the abstraction / responsibility of the data layer becomes blurred, as there are now multiple points of DB entry (each conversion retrieves records). Redundant code will still need to be written.
* **Option 3: Conversion runs on top of data retrieval**  
  The data is first retrieved from the database, before a conversion is made. The responsibility is very clear with this option, as data layer simply obtains data. However it is more complex to implement. There is much MUCH less abstraction, as the conversion and data layer need to have some agreed format (eg a record for a database-based design). Furthermore, redundant memory is used: a DB call is made to fetch data into memory, before it is converted into another structure in memory.

If this latter choice was to be made, **Option 1** is to be chosen, because there is a higher level of abstraction, single responsible class, and single point of entry for DB calls.

# Testing

Methods of testing

# Terminologies

Terminologies

* UI
  + **AP** (Assessment Panel): The far left UI component that shows the tree structure of courses and assessments.
  + **APNode**: The nodes in the tree view of the assessment panel UI
  + **PP** (Participant / Group Panel): The middle UI component that shows what participants exist for a particular course.
  + **PPNode**: The nodes in the tree view of the Participant / Group panel UI.
  + **VEP** (View / Edit Panel): The rightmost UI component that shows information based on what is selected in the PGP and VEP (refer to section 'Function Rules').
  + **EDF** (Edit Dialogue Form): This is the dialogue box that is responsible for taking in new changes of information that a user wishes to make to an assessment, component, student, or SMI.
* Entities
  + **Course**: A subject which contains multiple assessments. It helps group assessments together. VERY SIMILAR IN SEMANTICS to a group.
  + **Assessment**: An assessment task. It could be a quiz, or an assignment, or something else. It should be able to be split into components. Contains important information such as due date, weighting,
  + **Component**: Individual parts of an assessment that are worth marks. Differ based on the assessment type in real life. For quizzes or tests, a component is a question. For assignments, a component is a task / deliverable. A component can also be associated with another component - eg subquestions or sub parts.
  + **Group**: A collection of students. In real life, this could be a class or a team. Associated with an assessment. VERY SIMILAR in semantics to a course.
  + **Student**: An entity who is being marked, and is associated with a group. They may also have various SMI's. Information held: name, ID
  + **SMI**: Student Mark Information. Contains a student's mark and feedback FOR A LOWEST-LEVEL component.

# Questions

Feature requests:

* Feedback not ONLY for the student, but for the assessment? For the component? For the group?