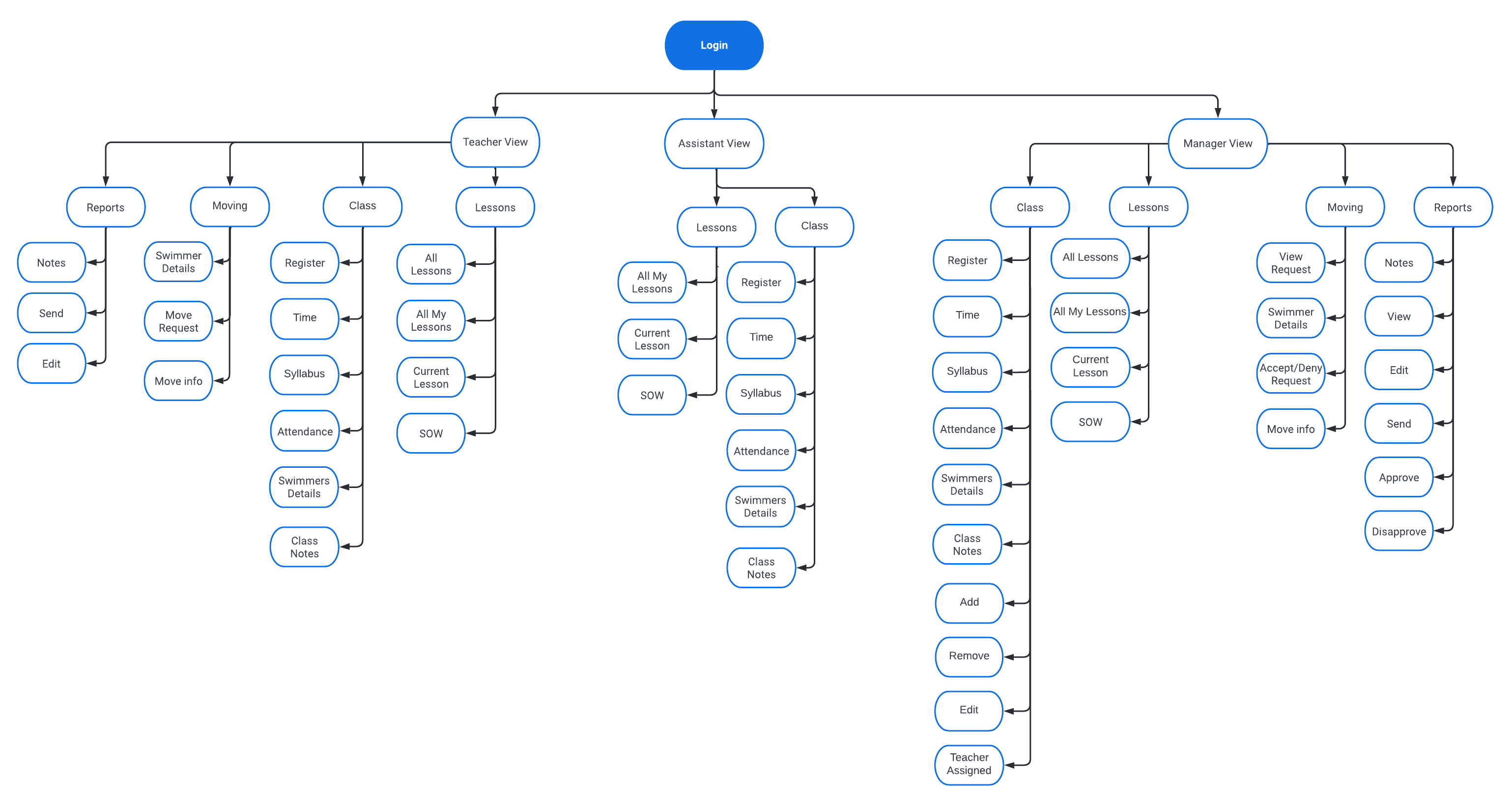
**Design**

In this Design chapter I will detail how I plan to build my ‘**Lesson Manager**’ system by using research made in my previous chapters. I will breakdown the overall system into sub-problems through the use of **Hierarchical Diagrams, Wireframes, Data Dictionaries, ERDs, DFDs and Flowcharts.** By doing this I will be able to separate the system into smaller parts which will all contribute to the final version.

By using **Hierarchical Diagrams,** such as a ‘Menu Diagram’, I will be able to visualise each of the modules within the system and assign functionalities to each module. I can then make amendments to any modules as I see fit.

**Data Dictionaries** and **Normalisation** allow for more data integrity as well as ensuring that all data has been assigned the correct data type and field length.

**ERDs** and **DFDs** aid by giving an overview of how all the parts of the database work together, as well as being able to identify primary and foreign keys for linking data.

**System Overview: Menu Diagram**

**Login module:** Allows the user access to the lesson manager through a 4-digit PIN – give the user the option to log into ‘**Teacher View’** or ‘**Manager View**’.

**Manager View module:** Allows the manager to view the whole lesson manager with all its functionalities. This has the highest user access level.

* **Lessons:** Managers can select how to view their lessons i.e., as ‘**All lessons’**, ‘**All my lessons’**, ‘**Current lesson’ or ‘SOW’.** ‘**SOW’** contains the goals for each level to achieve for that week.
* **Class:** Contains the information for teaching a class i.e., Level, Time, Swimmer details etc. The manager is also able to **Add** and **Remove** classes as well as **Edit** them. They can also access additional features through the **Class** function, such as the **Register and Syllabus**. They can also edit all class notes i.e., if any equipment is needed,
* **Moving:** This is where managers review move requests to see if the swimmer should move up/down or stay. This comes with ‘**Move info’** which contains a **justification for the request**. This contains information such as: swimmer’s **skillset**, **aptitude** for learning and any **other relevant points**. They can **‘Accept/Deny’** this request.
* **Reports:** This is where the manager writes the end of term progress for each group and emails it to their parents. The reports have access to the ‘**Notes’** for each individual swimmer to help with the report process. The manager views the report and then **decides whether it is ready** to be sent. When it is ready, they will ‘**Approve’** the report and come to an **overview of the report** before finally sending it to the parents. If they ‘**Disapprove’** the report, the **teacher becomes notified**, and the **report is removed** from the **manager’s view**. They will repeat this process for each class.

**Teacher View module:** Allows teachers to view an abstract version of the ‘Manager View’ with most permissions. This is the second highest user access level.

* **Lessons:** Same permissions as ‘Manager View’.
* **Class:** Same permissions as ‘Manager View’ **excluding** ability to **Add/Remove/Edit** classes.
* **Moving:** Here teachers can decide what swimmers they want to move up to the next level (or to keep them at the same level). This then gets submitted as a ‘**Move Request’** which notifies the manager of what swimmers to move and any additional ‘**Move info’.**
* **Reports:** Same functionalities as the manager, excluding the option to send it to the parents. Instead, the report gets sent to the manager for review.

**Assistant View module:** Allows assistants to view an abstract version of the ‘Teacher View’ with fewer permissions. This is the lowest user access level.

* **Lessons:** Same permissions as stated with Teachers. **Removed ‘All lessons’** view.
* **Class:** Same permissions as stated with Teachers.

**Normalisation**

The purpose of **Normalisation** in this project is to help develop a solution that will **reduce redundant data**, improve **data consistency**, **remove program data dependencies,** and improve the **security** of the data.

**3NF**

I have created a **3NF** to help create my final **ERD** for the system.

**Registers**(register\_ID, swimmer\_ID\*, lesson\_ID\*, attendance)

**Lessons**(lesson\_ID, class\_ID\*, week\_num\*)

**Swimmers**(swimmer\_ID, first\_name, surname, email, phone)

**Class**(class\_ID, teacher\_login\*, swimmer\_ID\*, level\_num\*, day, time)

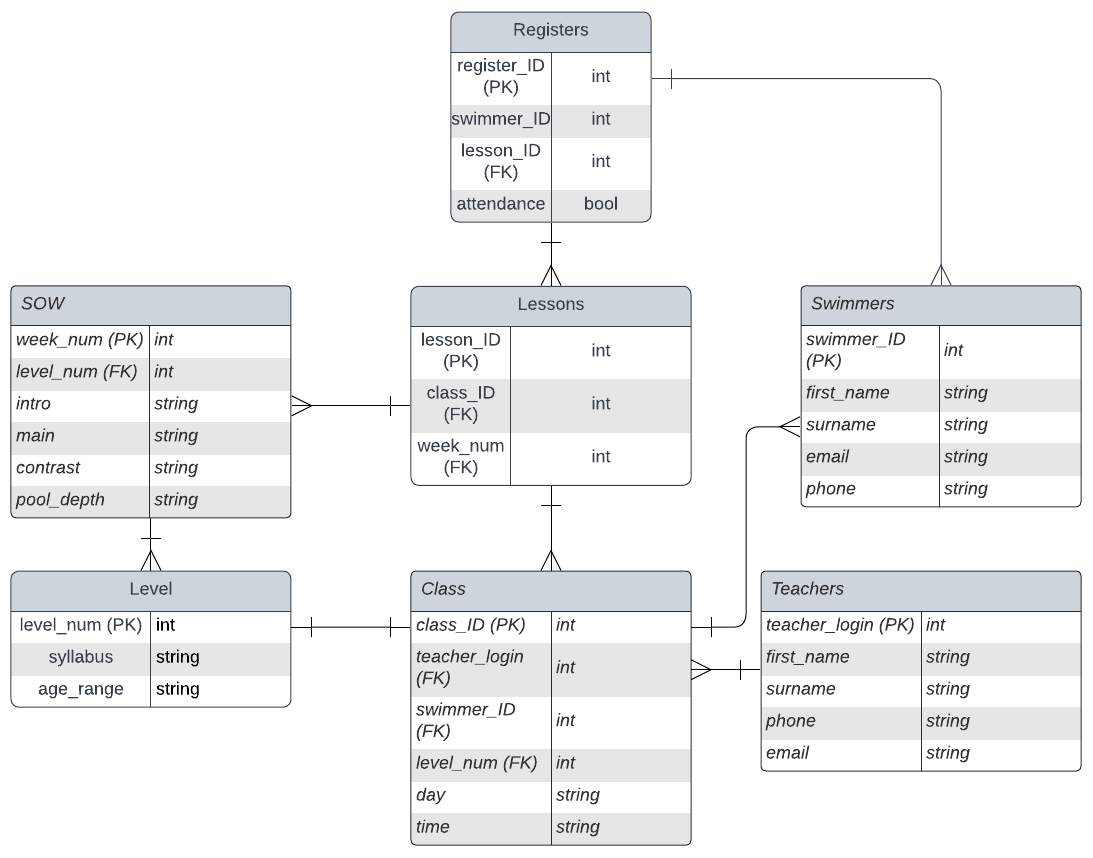
**Teachers**(teacher\_login, first\_name, surname, phone, email)

**SOW**(week\_num, level\_num\*, intro, main, contrast, pool\_depth)

**Level**(level\_num, syllabus, age\_range)

**ERD**

This is the final **ERD** for the new Register Database system:



**Data Dictionaries**

Data dictionaries make it easier to focus on individual pieces of data and how it should be implemented into the database. This is an important step of designing the system as we must ensure that the data stored will be recognised by the system in the correct way, thus avoiding any logic errors.

These following dictionaries will hold the data associated with each entity. These entities may change or be added to in correspondence with the club’s needs. This will not affect the program as the data will be independent from the programming.

**Class**

The class table relating to each individual class within the system. When the manager adds/edits a class in the system this is the data they will be changing. The information stored in this table will also be the information displayed in the system to give the user an idea of the class they are about to teach.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Item | Data type | Format | Bytes for storage | Description | Example | Validation |
| class\_ID (PK) | Integer | NN | 2 bytes | Unique identifier for ‘class’ | 01 | n/a |
| teacher\_ID (FK) | Integer | NN | 2 bytes | Unique identifier for ‘teachers’ | 12 | format check |
| swimmer\_ID (FK) | Integer | NN | 2 bytes | Unique identifier for ‘swimmers’ | 17 | format check |
| level\_num (FK) | Integer | N | 1 byte | Unique identifier for ‘level’ | 7 | Length check |
| time | string | NN:NN | 5 bytes | The time for the class | 17:30 | Format check |

**Lessons**

This table will be able to store all classes and the week for those classes. The idea behind this is that whatever week it is, only those classes will be selected. This is solely for creating joins between different tables in the database.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Item | Data type | Format | Bytes for storage | Description | Example | Validation |
| Lesson\_ID (PK) | integer | NN | 2 bytes | Unique identifier for ‘lessons’ | 23 | n/a |
| Class\_ID (FK) | integer | NN | 2 bytes | Unique identifier for ‘class’ | 12 | n/a |
| Week\_num (FK) | integer | N | 1 byte | Unique identifier for ‘SOW’ | 7 | Length check |

**Register**

This table will store the attendance of each swimmer in each lesson. The user will be able to change the swimmers’ attendance by pressing a button, which will change the **Boolean state** of attendance.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Item | Data type | Format | Bytes for storage | Description | Example | Validation |
| Register\_ID (PK) | integer | NN | 2 bytes | Unique identifier for “Registers” | 10 | n/a |
| Swimmer\_ID (FK) | integer | NN | 2 bytes | Unique identifier for ‘swimmers’ | 12 | n/a |
| Lesson\_ID (FK) | integer | NN | 2 bytes | Unique identifier for ‘lessons’ | 23 | n/a |
| attendance | bool | N | 1 byte | 0 = Absent whereas  1= present | 0 | Format check |

**Swimmers**

All information regarding swimmers will be stored here. If a teacher would need to get a hold of the swimmer’s parents, they can view their contact information.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Item | Data type | Format | Bytes for storage | Description | Example | Validation |
| Swimmer\_ID (PK) | integer | NN | 2 bytes | Unique identifier for “Swimmers” | 10 | n/a |
| first\_name | string | NNNNNNNNNN | 10 bytes | Stores the swimmer’s first name | “Daniel” | Type check |
| surname | string | NNNNN… | 12 bytes | Stores swimmer’s surname | “Vachagan” | Type check |
| phone | string | NNNNNNNNNNN | 11 byte | Stores the **parents’** phone number for emergency contact | “07968444333” | Type check/  length check |
| Email | string | NNNNNN@NNNNNN.com | 32 bytes | Stores **parents’** email for sending progress | “db@gmail.com” | Format check |

**Teachers**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Item | Data type | Format | Bytes for Storage | Description | Example | Validation |
| Teacher\_login  (PK) | integer | N | 1 byte | Unique identifier for “Teachers” | 1 | n/a |
| First\_name | string | NNNNNN | 10 bytes | Stores teacher’s first name | Frleece | Type check |
| Last\_name | string | NNNNNN | 12 bytes | Store teacher’s last name | Shekane | Type Check |
| phone | string | NNNNNNNNNNNN | 11 bytes | Teacher’s personal phone no. | 07999444555 | Length check/  format check |
| email | string | NNN@NNNNN.com | 32 bytes | Teacher’s personal email | db@gmail.com | Format check |

**Level**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Item | Data type | Format | Bytes for Storage | Description | Example | Validation |
| Level\_num (PK) | integer | N | 1 byte | Unique identifier for “Level” | 1 | n/a |
| syllabus | string | NNNN…  (however long is necessary) | 1 kilobyte | Contains info on the skills each swimmer should achieve for that level | Be able to swim for 25m without aid | Length check/  presence check |
| Age\_range | string | NN-NN | 5 bytes | The age range for that level | 10-12 | Presence check |

**SOW**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Item | Data type | Format | Bytes for storage | Description | Example | Validation |
| Week\_num (PK) | integer | N | 1 byte | Unique I dentifier for ‘SOW’. Only goes up to week 7. | 7 | n/a |
| Level\_num (FK) | integer | N | 1 byte | Unique identifier for ‘Level’ | 3 | n/a |
| Intro | string | {text} | ½ kilobyte | Info for the ‘introduction’ set | Get in the water | Presence check |
| Main | string | {text} | ½ kilobyte | Info for the ‘main’ set | Swim 25m with float front/back | Presence check |
| Contrast | string | {text} | ½ kilobyte | Info for the ‘contrast’ set | Dive through hoops underwater | Presence check |
| Pool\_depth | string | N.Nm | 4 bytes | Info for the required depth of the pool | 1.2m | Presence check/  format check |

A white circle with black text

Description automatically generated**Level 0 DFD**

This is the **context diagram** for the system.

This shows **how** the **data is transferred** throughout the system and how it **interacts** with the **database**.

**Wireframes**

A screenshot of a computer

Description automatically generated**Staff Selection**

Here the user will decide what to login as. Depending on the role they select, they will be assigned an access level.

**Pseudocode:**

IF "Manager Login" SELECTED:

    access\_level = 0

selected\_role = “Manager”

ELIF "Teacher Login" SELECTED:

    access\_level = 1

selected\_role = “Teacher”

ELIF "Assistant Login" SELECTED:

    access\_level = 2

selected\_role = “Manager”

A screenshot of a computer

Description automatically generated**Login Screen**

These boxes change colour to signify a digit has been input in that slot. Doesn't show digit in the slot for security.

**Pseudocode:**

connect to myDataBase as myDB

all\_logins = "SELECT Logins FROM Staff"

IF ENTERED\_LOGIN is in all\_logins:

    PROCEED to Lesson Manager

'Clears' the input for the PIN. Changes colours of boxes back to neutral state.

Submits the login request with the input PIN. Returns error for criteria not met for the input.

**Main Menu**

A screenshot of a computer

Description automatically generated

This is where the user can select **how to view all classes** as well as **reports.** They can also view the **SOW** for the week from this menu.

**Lesson Screen**

A screenshot of a computer

Description automatically generated

**Pseudocode:**

connect to myDataBase as myDB

selected\_day = {Button Pressed}

Classes = "SELECT ALL\_INFO FROM Classes WHERE Day={selected\_day}"

FOR ClassID in Classes:

    class\_register = Button.grid(value = {ClassID})

FOR StaffID in Classes:

    Get Teacher name

    class\_teacher = Label.grid(text = {Teacher name})

FOR Time in Classes:

    class\_time = Label.grid(text = {Time})

FOR SOWID in Classes:

    class\_sow = Button.grid(value = {SOWID})

The title for this screen will change depending on what the user selects in the **Main Menu**. Only one class is shown for the sake of this example, but for more classes a scrollbar has been implemented.

**Registry**

The button is set to ABSENT by default. Changes text and colour on each click i.e. **PRESENT** and **ABSENT**.

A screenshot of a computer

Description automatically generated

The **MARK** button allows the Teacher to assess swimmers on their swimming skills through the registry.

**Assessing Swimmers**

A screenshot of a computer

Description automatically generated

Each button has 2 states:

* **FAIL**
* **PASS**

All buttons will be set to **FAIL** by default.

**Pseudocode:**

IF Button pressed:

Change text and colour

    AND

    UPDATE myDB WHERE Name = **Swimmers Name**

This will allow the Teacher to keep track of the skills that a swimmer has achieved in their level. If the Teacher wishes to move the swimmer **UP/DOWN** a level, they can select the **MOVE** button.

**Moving Swimmers**

A screenshot of a computer

Description automatically generated

The purpose of moving swimmers is to place them in a different level of class.

Here the teacher will select **what class the swimmer should move to**. This view will be sorted by **LEVEL NO**. (lowest to highest).

Doing so will **change the swimmer’s class\_ID**, thus removing them from their initial class.

A screen shot of a confirmation

Description automatically generated

This pop up will show to **ensure** that there is **no mistake** in moving the swimmer up/down a level.

**Pseudocode:**

connect to myDataBase as myDB

WHEN new\_class IS SELECTED:

    make SAVE\_CHANGES button active

    new\_class\_id = new\_class.get(ClassID)

WHEN SAVE\_CHANGES IS PRESSED:

    CREATE messagebox(text="This swimmer is moving from {current level} to {selected\_level} YES/NO")

    IF "YES":

        "UPDATE ClassID of Swimmer to new\_class\_id"

    ELSE:

        return

A diagram of a manager

Description automatically generated**SOW**

On clicking the button, all the info for 'intro', 'main', 'contrast' and 'depth' will be retrieved from the database and displayed into each of these boxes.

**Pseudocode:**

connect to myDataBase as myDB

WHEN new\_level button IS PRESSED:

    new\_sow = "SELECT Intro, Main, Contrast, Depth FROM SOW WHERE level={new\_level}"

FOR each\_sow in new\_sow:

    INSERT each\_sow into TEXTBOX

**Blue** highlight on the button shows that it has been **selected**. **Grey** means it **hasn't**.

**Editing Classes**

A screen shot of a computer screen

Description automatically generated

Managers will be able to use the ‘**Edit Class’** button in the **Lesson Screen** which will bring up the **SOW** and **list of swimmers**.

Here they can **remove swimmers** and **edit the SOW** as well as saving their changes.

Being able to add swimmers here was too complex which is why I only made the **REMOVE** function.

Adding swimmers is a different function found in the **LESSON SCREEN.**

**Pseudocode:**

connect to myDataBase as myDB

WHEN Edit\_Class button IS PRESSED:

    GET ClassID and SOW

    all\_swimmers = "SELECT ALL\_SWIMMERS FROM Swimmers WHERE class\_id={ClassID}"

treeview(values = all\_swimmers)

treeview.grid()

WHEN swimmer in treeview IS SELECTED:

    make REMOVE button active

IF REMOVE button IS PRESSED:

    swimmer\_id = GET swimmer.SwimmerID

    "UPDATE Swimmers SET ClassID=0 WHERE SwimmerID={swimmer\_id}"

A screenshot of a computer

Description automatically generated**Viewing Swimmers**

Here the manager will be able to view all swimmers and even change their details such as **Personal Info**, **Classes** and **Notes**.

Managers will also be able to **ADD** new swimmers to the database by entering the swimmer’s details into the **Swimmer Details** box and pressing **ADD.**

The selected swimmer will be highlighted as well as populate the **Swimmer Details** with their info.

**ADD** is only available to Managers. **When** **disabled** it will be **greyed out** as shown.

**CLEAR** empties all the current **Swimmer Details**.

**Pseudocode:**

connect to myDataBase as myDB

IF access\_level = 0:

    make ADD button active

all\_swimmers = "SELECT ALL\_SWIMMERS FROM Swimmers"

treeview(values = all\_swimmers)

treeview.grid()

CREATE entry\_widgets FOR all\_swimmers

WHEN swimmer in treeview IS SELECTED:

    GET swimmer\_id

    populate entry\_widgets with respective swimmer info

WHEN CLEAR button IS PRESSED:

    entry\_widgets.clear()

WHEN SAVE button IS PRESSED:

    "UPDATE swimmer WHERE SwimmerID={swimmer\_id}"

WHEN ADD button IS PRESSED:

    swimmer\_details = entry\_widgets.get()

    "INSERT new\_swimmer INTO Swimmers VALUES={swimmer\_details}"

A screenshot of a computer screen

Description automatically generated**View Classes**

The **Manager** will be able to view all classes as well as change certain details of the classes. For example, when a class is **selected it is highlighted in blue** and the **Class Details** box will be populated with its info. From there, we can change anything from **Assigned Staff, Level, Day, Time.**

We can also sort how we view each all the classes by **Level**, **Time** and **Day.** This will make **finding classes easier** for the Manager.

Adding classes will also be available through here.

**Pseudocode:**

connect to myDataBase as myDB

GET ClassID

all\_classes = "SELECT ALL\_CLASSES FROM Classes"

treeview(values = all\_classes)

treeview.grid()

CREATE combo\_boxes FOR all\_classes

WHEN class in treeview IS SELECTED:

    SET combo\_boxes to respective class info

WHEN SAVE button IS PRESSED:

    "UPDATE class WHERE class\_id={ClassID}"

WHEN ADD button IS PRESSED:

    class\_details = combo\_boxes.get()

    "INSERT new\_class INTO Classes VALUES={class\_details}"

CREATE sort\_by\_boxes

WHEN sort\_by\_boxes ARE CHANGED:

    sort by whatever sort\_box was changed

A screenshot of a computer

Description automatically generated**View Staff**

The Manager can **view all staff** part of the system and **change their info** as well.

From here we can also choose to add new staff to the system and assign them a role as **Manager**, **Teacher** or **Assistant**.

**Pseudocode:**

connect to myDataBase as myDB

IF access\_level = 0:

    make ADD button active

all\_staff = "SELECT ALL\_STAFF FROM Staff"

treeview(values = all\_staff)

treeview.grid()

CREATE entry\_widgets FOR all\_staff

WHEN staff in treeview IS SELECTED:

    GET staff\_id

    populate entry\_widgets with respective staff info

WHEN CLEAR button IS PRESSED:

    entry\_widgets.clear()

WHEN SAVE button IS PRESSED:

    "UPDATE staff WHERE StaffID={staff\_id}"

WHEN ADD button IS PRESSED:

    staff\_details = entry\_widgets.get()

    "INSERT new\_staff INTO Staff VALUES={staff\_details}"

A screenshot of a computer

Description automatically generated**Reports**

**Teachers** and **Managers** will have access to reports. Both can select any swimmer within the system and write up a report about the swimmer regarding their performance.

Once a report is written up, they can decide to **save** the report for **reviewing** later OR **send the report** off to the email connected to the swimmer.

**Pseudocode:**

connect to myDataBase as myDB

all\_swimmers = "SELECT ALL\_SWIMMERS FROM Swimmers"

treeview(values = all\_swimmers)

treeview.grid()

CREATE ReportBox and make it disabled

WHEN swimmer IS SELECTED:

    GET SwimmerID

    make ReportBox active

    make SEND\_REPORT button active

WHEN SAVE button IS PRESSED:

    report\_contents = ReportBox.get()

    "UPDATE Swimmer MAKE Report={report\_contents} WHERE swimmer\_id={SwimmerID}"

WHEN SEND\_REPORT button IS PRESSED:

    email = "SELECT Email FROM Swimmers WHERE swimmer\_id={SwimmerID}"

    send report to email

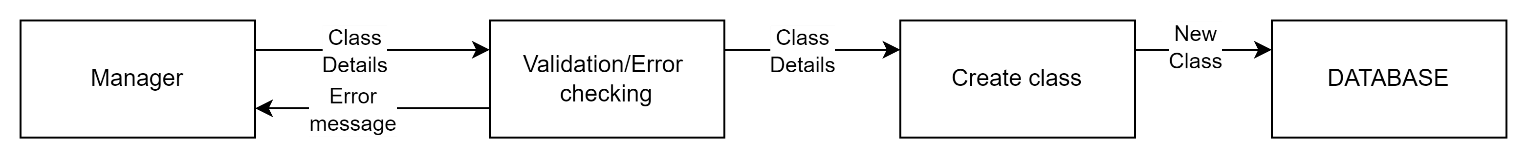
A screenshot of a computer

Description automatically generated**Level 1 DFD** **– Login**

This is a Level 1 DFD for Logging into the system.

Each member of staff has a role which they are assigned as they are added to the system. Upon selecting what access level they have in **Staff Selection.** If the **PIN** in the **DATABASE** matches with the corresponding access level, then that access level is passed on into the main program.

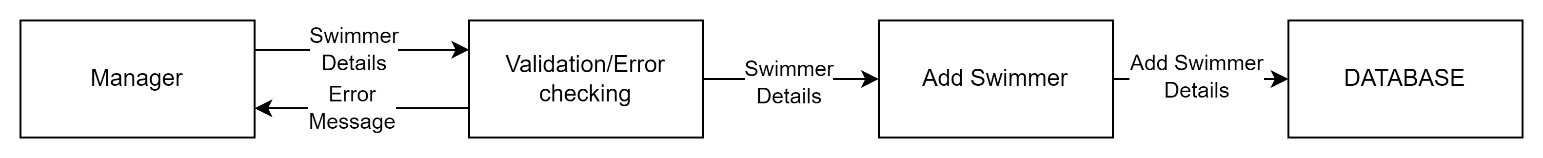
**Level 1 DFD – Adding Classes**



**Level 1 DFD – Adding Staff**



**Level 1 DFD – Adding Swimmers**



**Level 1 DFD – Updating Classes**



**Level 1 DFD – Making/Sending Reports**

A black background with white rectangles

Description automatically generated

**Flowcharts**

**Validation**

**Validation** is import within any robust system as it helps to **prevent logic errors** and **crashes**. By having Validation, it helps both the programmer and the user understand what is happening in the system and what actions need to happen. This will overall reduce time wasted on trying to fix problems in my system.

To help implement this, I have designed **flowcharts** which will be **a guide to designing** the **Validation/Verification** for the system.

**Presence Check**

This type of check will ensure that an input is entered when needed and if there isn’t, the program will not crash. Instead, it will simply return an **error message** to the user, asking for an input.

A screenshot of a computer screen

Description automatically generated

**Type Check**

This type of check will ensure that the user enters the correct data type i.e. **Integer** OR **String**. If not, it will return an **error message** to the user, asking for the **desired data type**.

A group of white squares with black text

Description automatically generated

**Format Check**

A screenshot of a computer screen

Description automatically generatedThis type of check will ensure that certain symbols are present where needed i.e. an email needs an **@ symbol**. By doing this, we can have a way of **Validating email addresses** within the system.

**Length Check**

This type of check will ensure that certain inputs maintain a set length. An important example of this would be Phone no. within the system. A Phone no. can be considered invalid if there is not **exactly 11 numbers.** This will also prevent more data being stored than is necessary in the system i.e. **Notes** OR **SOW**.

A screenshot of a computer screen

Description automatically generatedA screenshot of a phone

Description automatically generated