

Design

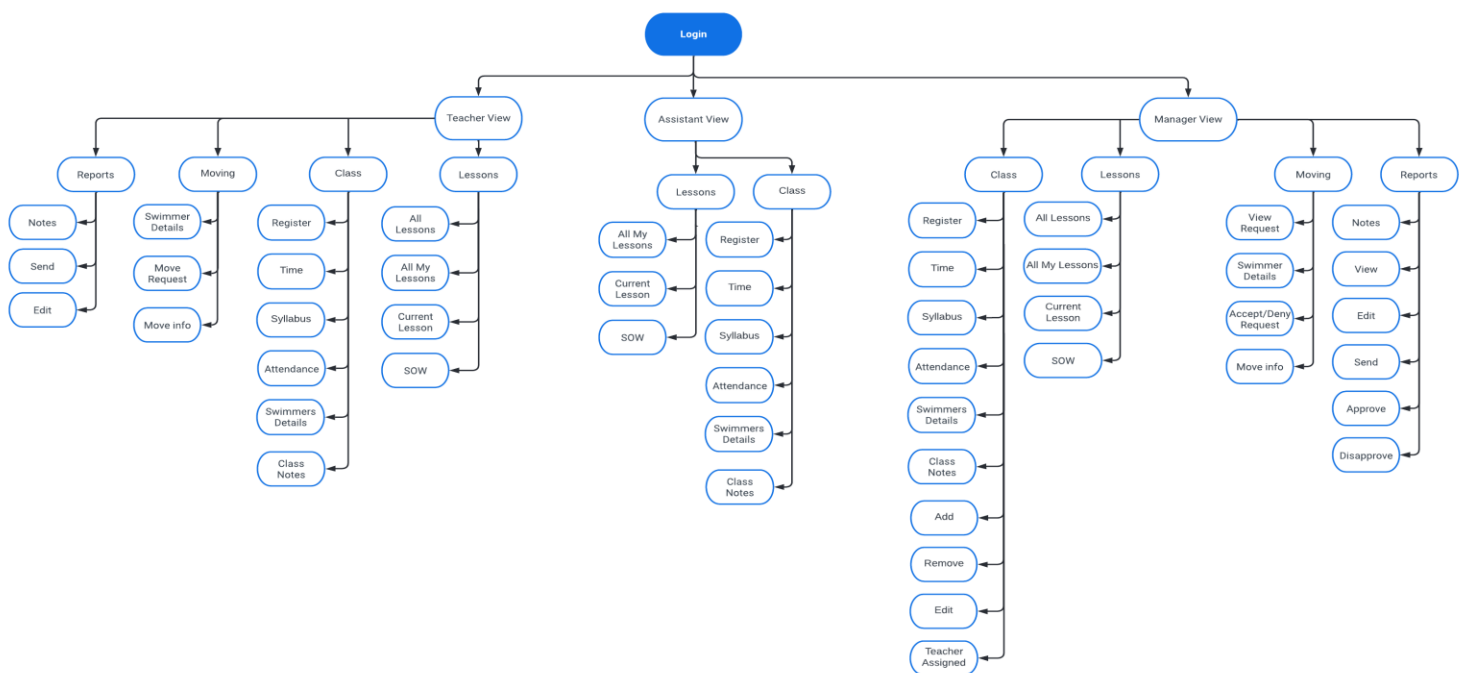
In this Design chapter I will detail how I plan to build my '**Lesson Manager**' system by using research made in my previous chapters. I will breakdown the overall system into sub-problems through the use of **Hierarchical Diagrams, Wireframes, Data Dictionaries, ERDs, DFDs and Flowcharts**. By doing 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)

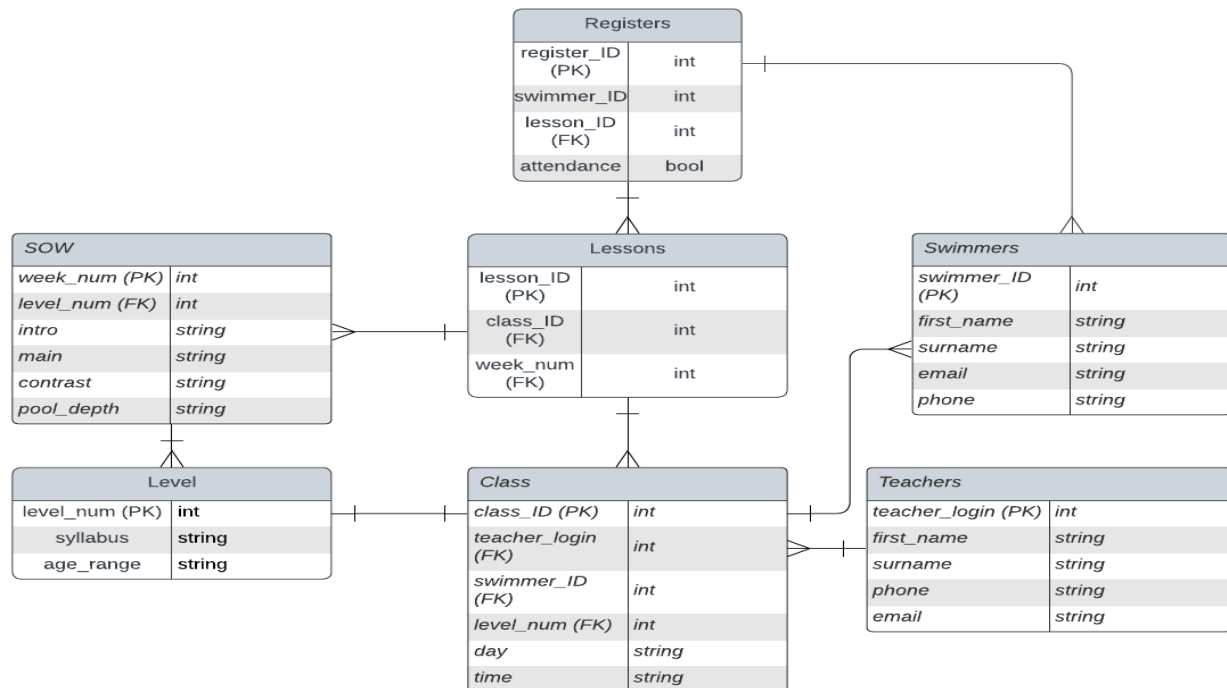
Teachers(teacher_login, first_name, surname, phone, email)

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	NNNNNN NNNN	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	NNNNNN NNNNN	11 byte	Stores the parents' phone number for emergency contact	"07968444333"	Type check/ length check
Email	string	NNNNNN @NNNNN N.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_log in (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

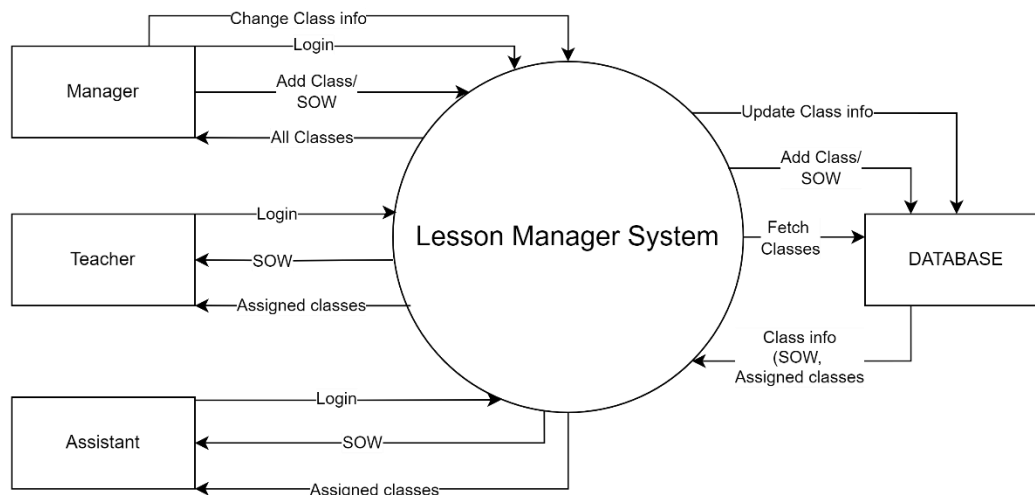
Level

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 identifier 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

Level 0 DFD

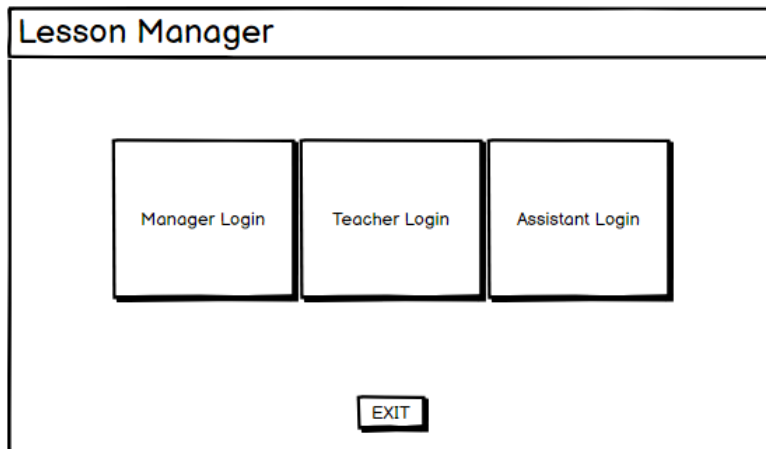


This is the **context diagram** for the system.

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

Wireframes

Staff Selection



The wireframe shows a rectangular window titled "Lesson Manager". Inside the window, there are three rectangular buttons arranged horizontally, labeled "Manager Login", "Teacher Login", and "Assistant Login". Below these buttons, centered at the bottom of the window, is a small rectangular button labeled "EXIT".

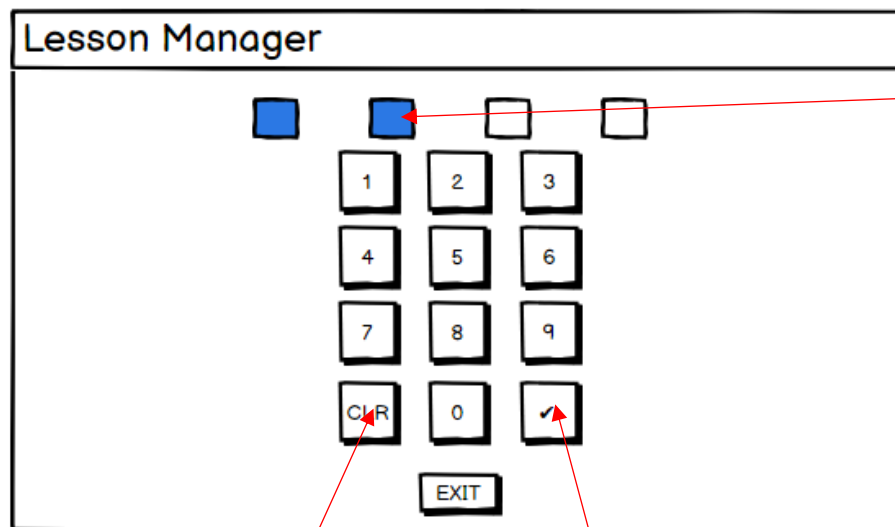
Here the user will decide what to login as.

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"
```

Login Screen



The wireframe shows a rectangular window titled "Lesson Manager". Inside the window, there are four empty square boxes at the top for displaying digits. Below these is a numeric keypad with buttons for digits 1 through 9, 0, a "CLR" button, and a checkmark button. At the bottom center is an "EXIT" button. Red arrows point from the text descriptions to the CLR, checkmark, and the top digit boxes.

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

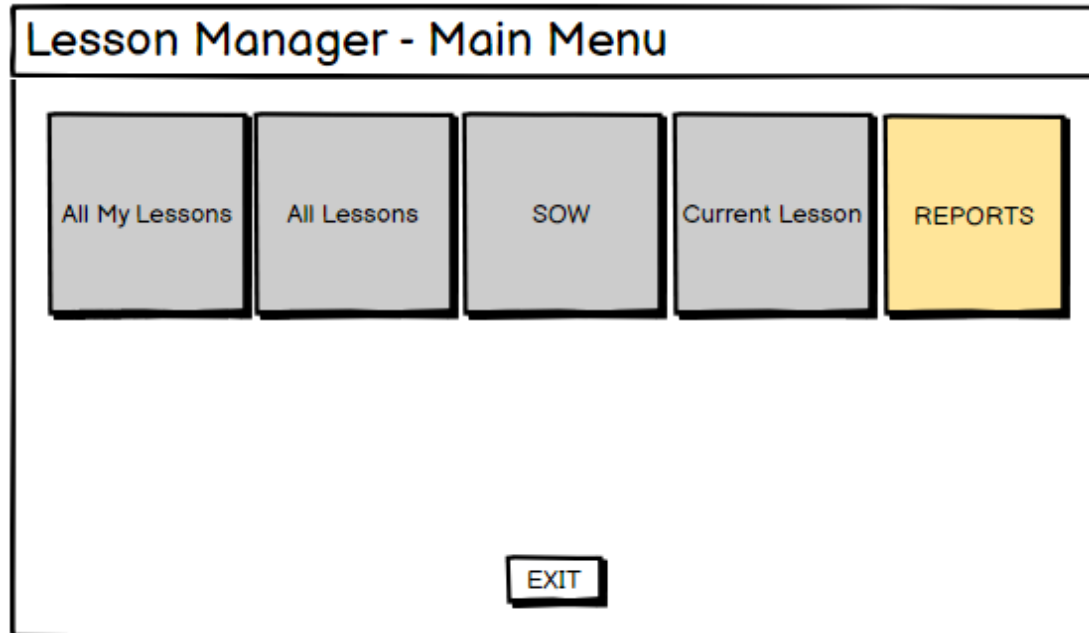
Pseudocode:

```
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.

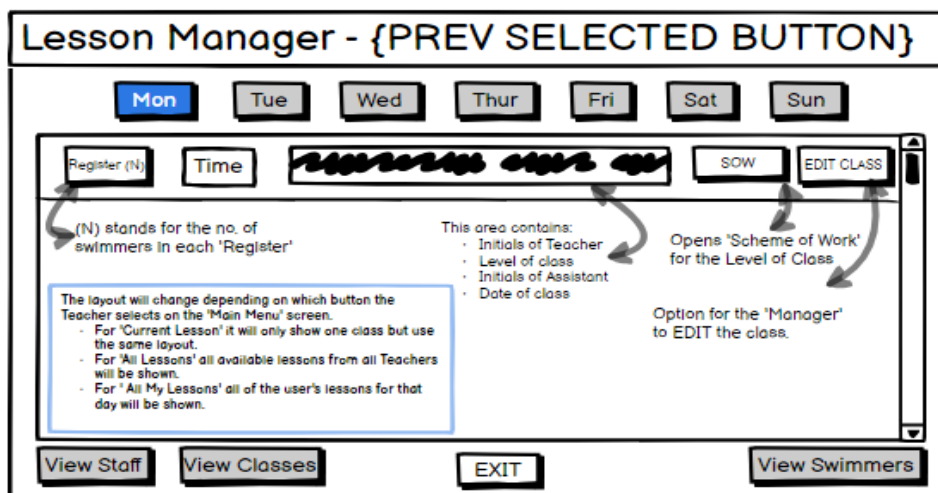
Submits the login request with the input PIN. Returns error for criteria not met for the input.

Main Menu



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



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.

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})
```

Registry

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

Lesson Manager - Register

Swimmers	Attendance	Assess
Name	Absent	Mark
Name	Absent	Mark
Name	Present	Mark

Here there would be a column of names each with their own attendance shown as above. This design would just be replicated for the number of students in the class.

EXIT

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

Assessing Swimmers

Lesson Manager - Assessment

{Swimmer's name} {level no.}

Syllabus

Skill	Assessment
{SKILL NO.1}	FAIL
{SKILL NO.2}	PASS
{SKILL NO.3}	FAIL
{SKILL NO.4}	FAIL

MOVE

EXIT

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

Lesson Manager - Moving

{Swimmer's name} {level no.}

Class_ID	Teacher (staff_ID)	Assistant (staff_ID)	Level No.	Time	Day
1	Dylan Branda (1)	n/a	Level 3	5:00	Monday
2	Dillon Corns (4)	Matthew Jordino (5)	Level 4	5:30	Monday
3	Peter Dave (2)	Matthew Jordino (5)	Level 5	5:00	Monday
4	Peter Dave (2)	n/a (5)	Level 5	5:30	Monday

SAVE CHANGES

EXIT

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.

Confirmation

{Swimmer's name} is moving from {current level} to {selected level}?

YES

NO

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
```

SOW

Lesson Manager - Scheme of Work

SOW - Week (n)

Level 1 Level 2 Level 3 Level 4 Level 5 Level 6 Level 7

Introduction	Main	Contrast	Depth
.....	0.5m - 2.0m

EXIT

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

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
```

Editing Classes

Lesson Manager - Edit Class

Introduction	Main	Contrast	Depth
.....	0.5m - 2.0m

Swimmer ID	Class ID	FirstName	LastName	Email	Phone
1	1	1111111111
.....

REMOVE SAVE

EXIT

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}"
```

Viewing Swimmers

Lesson Manager - All Swimmers

Swimmer ID	Class ID	FirstName	LastName	Phone	Email	Notes
1	1	Tom	Ford	07986543212	tf@gmail.com	Has asthma
2	1	Dylan	Branda	07888666555	db@gmail.com	N/A
3	1	Peter	Kelly	07888444333	pk@gmail.com	Deaf in one ear
4	2	Matthew	Tom	07999444999	mt@gmail.com	N/A

Swimmer Details

First Name

Phone

classID

{current classID}

Last Name

Email

NOTES

ADD

SAVE

EXIT

CLEAR

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

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

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.

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}"
```

View Classes

Lesson Manager - All Classes

Class_ID	Teacher (staff_ID)	Assistant (staff_ID)	Level No.	Time	Day
1	Dylan Branda (1)	n/a	Level 1	5:00	Monday
2	Dillon Corns (4)	Matthew Jordino (5)	Level 1	5:30	Monday
3	Peter Dave (2)	Matthew Jordino (5)	Level 1	5:00	Monday

Class Details

Teacher

{Select Teacher} ▾

Level

{Select Level} ▾

Time Slot

{Select Time} ▾

Assistant

{Select Assistant} ▾

Day

{Select Day} ▾

ADD CLASS

Sort By

Level

{Level} ▾

Time

{Time} ▾

Day

{Day} ▾

SAVE

EXIT

Adding classes will also be available through here.

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.

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
```


View Staff

Lesson Manager - All Staff

Teacher_ID	FirstName	LastName	Phone	Email	Pin	Role
1	Dylan	Branda	07999444555	db@gmail.com	7777	Manager
2	Bobby	Shmurner	07888444666	ohdeymus@gmail.com	6969	Assistant
3	Pieter	Kelly	07222111333	pk@gmail.com	9999	Teacher
4	Dillion	Corns	07666333222	dc@gmail.com	6666	Teacher

Staff Details

First Name

Phone

PIN

Last Name

Email

Role

{Current Role} ▾

ADD

SAVE

EXIT

CLEAR

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}"
```

Reports

Lesson Manager - Reports

Swimmer_I	Class_I	FirstNam	LastNam	Phone	Email	Notes
1	1	Tom	Ford	0798654321	tf@gmail.co	Has asthma
2	1	Dylan	Branda	0788866655	db@gmail.co	N/A
3	1	Peter	Kelly	0788844433	pk@gmail.co	Deaf in one e

Report


SAVE

EXIT

SEND REPORT

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.

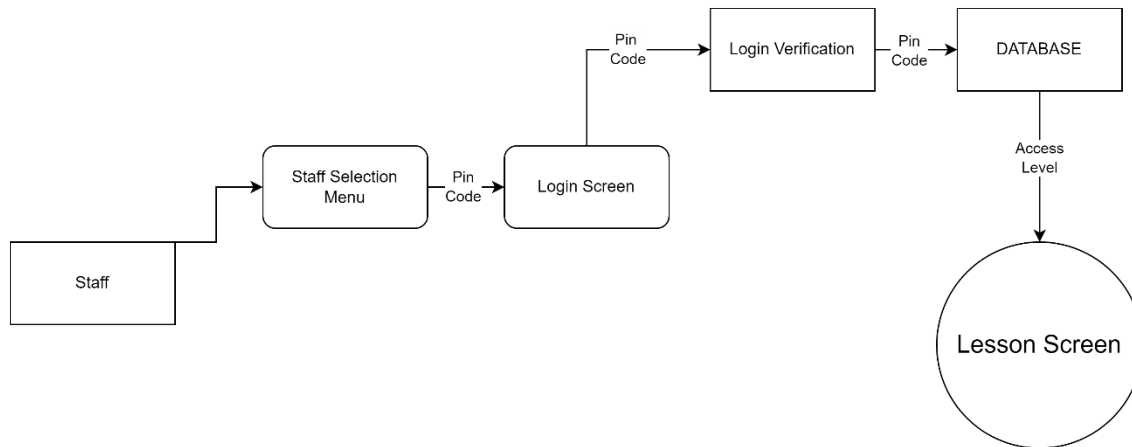
Once a report is written up, they can decide to **save** the report for **reviewing** later OR **send the report** off to the email connected to the swimmer.

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
```

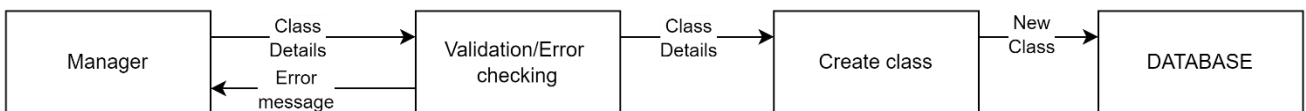
Level 1 DFD – Login



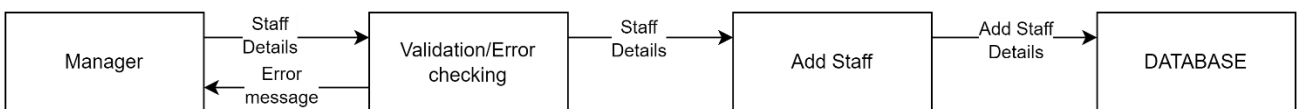
This is a Level 1 DFD for Logging into the system.

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.

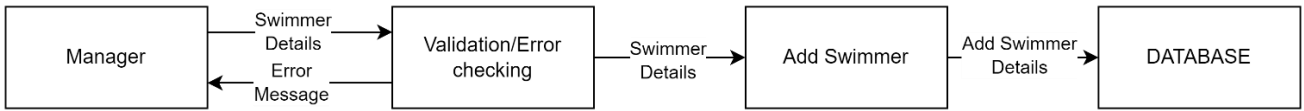
Level 1 DFD – Adding Classes



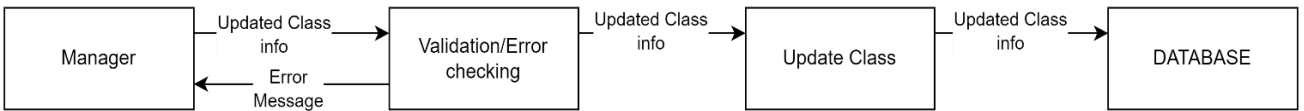
Level 1 DFD – Adding Staff



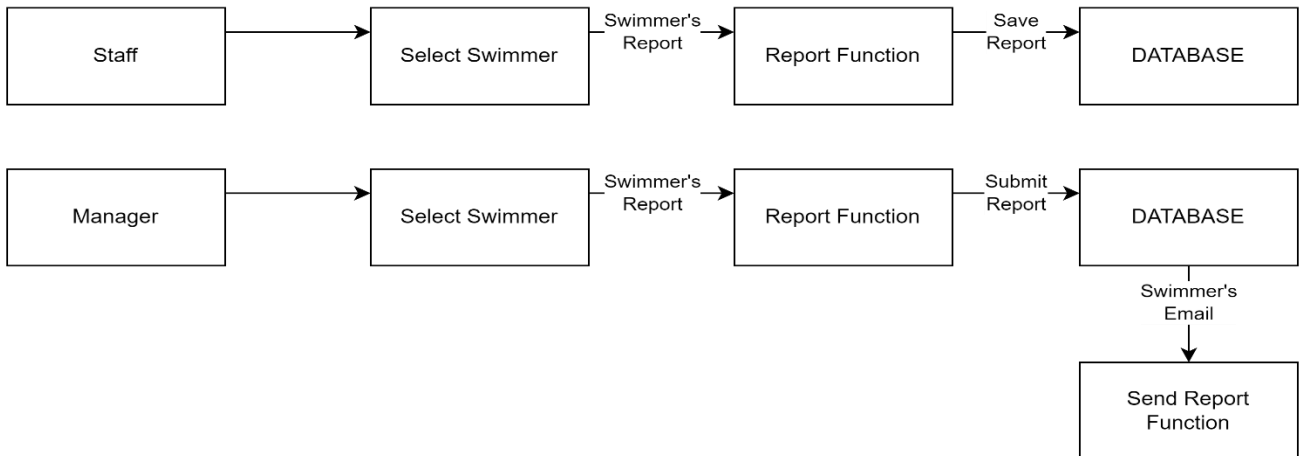
Level 1 DFD – Adding Swimmers



Level 1 DFD – Updating Classes



Level 1 DFD – Making/Sending Reports



Flowcharts

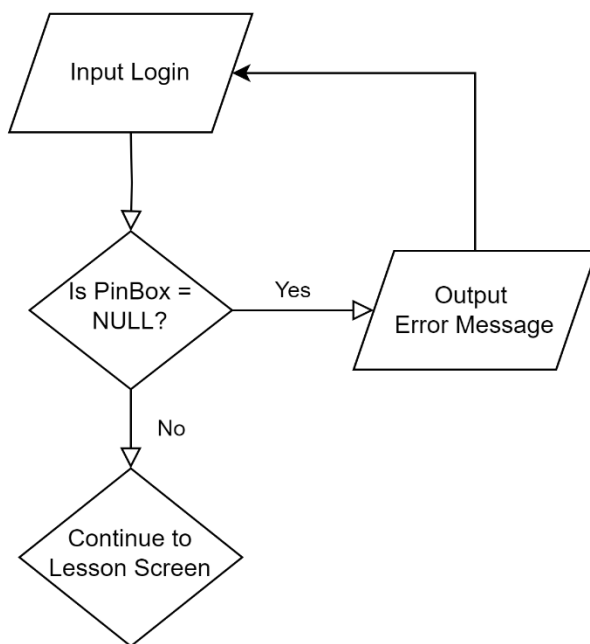
Validation

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.

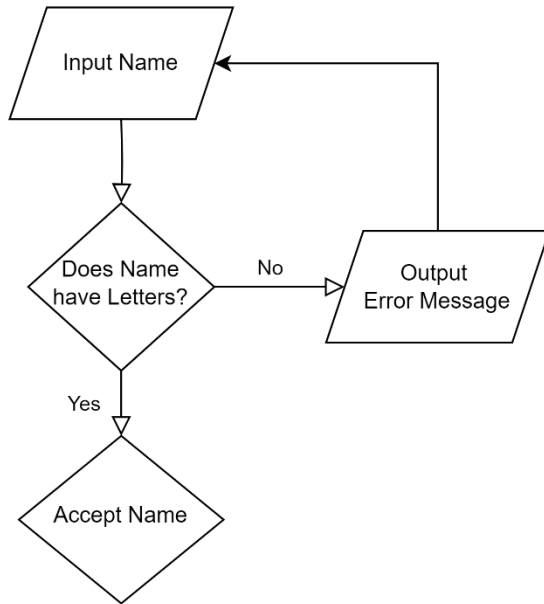
Presence Check

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.



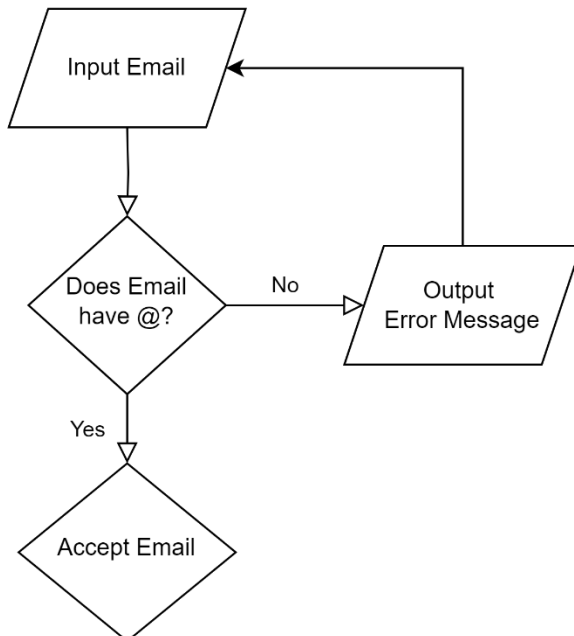
Type Check

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**.



Format Check

This 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**.

