# International Baccalaureate Diploma Programme

# Computer Science

# Internal Assessment – Computational Solution

# Table of Contents

# Criterion A – Problem Specification

## ***Problem Scenario***

An IB World School attended by a friend of mine tracks which DP courses their students take by manually updating a digital document every time new students arrive and/or every time that a course is changed. Since the school creates a timetable of classes based on this document by manually searching which student takes which course, this poses the school a large inconvenience. This is because the timetables created are prone to errors and clashes, and are inconvenient to both students and teachers, I have decided to find a solution to this specific problem.

I have decided to create a software solution that may fix this issue for any school facing it. This software application should be able to allow for coordinators and/or other heads of schools to input which courses are offered by the IBWS, together with information about students, along with which courses they attend, with that data being stored persistently, and with an option that would lead to the automatic creation of a balanced timetable that would fit the needs of both students and educators.

## ***Computational Context***

To develop my application, I have chosen to use Java for the following reasons:

- Java runs on the JVM, which will allow the application's single codebase to be platform-independent, and it allows for collaboration between different JVM languages if needed.
- Java's syntax is very legible, easy to use, and easy to learn.
- Java is an object-oriented language, which will be especially useful to store and serialize data inside my application since students could be stored as objects defined by a class.
- Java is a popular language with a rich ecosystem of libraries, some of which I could use to develop GUIs, features, etc.
- I could use Java's OOP features and its standard library to serealize data about students and subjects, as well as sort them for the timetable, and easily retreive them from the database.

To store data persistently, I have decided to use MySQL because:

- It is one of the most popular databases, featuring lots of useful user documentation.
- It has a simple yet powerful syntax.
- It stores data using tables, which will be especially useful for storing data about students and which courses they attend.
- Java features MySQL in its standard library, allowing me to use both without any additional necessities.
- Due to the simple nature of the solution, it would be unnecessary to use more complex databases, so, to make the process simpler for both myself and the target audience which might use the app, the most basic functionalities and queries of MySQL can be used.

***Success Criteria***

1. The app will allow the user to input basic data about a student.
2. The app will allow a user to state which 6 IBDP courses a student takes.
3. The app will persistently store all data about a student, including courses.
4. The app will be able to automatically create a timetable of classes based on given data.
5. The timetable will not feature clashes of two classes.
6. Classes taken by fewer students will be put at the front of the schedule so that most students and teachers do not have to arrive at school earlier than necessary.
7. The app user will be able to input which courses are offered by the school using a configuration file, as well as how many times classes in a week that subject features.

# Criterion B – Planning

## ***Decomposition***

1. Project identification and specification
   - Identify a real-world issue
   - Outline if and how the issue can be solved using a computational solution
   - Outline the solution idea

2. Project planning and design
   - Research and choose programming and query language
   - Identify the solution success criteria
   - Sketch and draft the software concept and design
   - Define the solution overview, design, and organization

3. Project setup
   - Set up the Java project in the IDE
   - Set up the MySQL database

4. Backend features development
   - Construct a class defining all fields necessary for a student to have
   - Construct a file reading system to determine which courses the school offers
   - Construct methods needed to update and modify the database from the Java app

5. Frontend features development
   - Construct an app GUI
   - Allow for the creation of new students via the GUI
   - Construct an algorithm for automatic balanced timetable creation

6. Testing, fixing and evaluation
   - Personally test the product
   - Fix any found bugs
   - Beta testing with a wider audience
   - Fixing bugs and improving user experience based on feedback

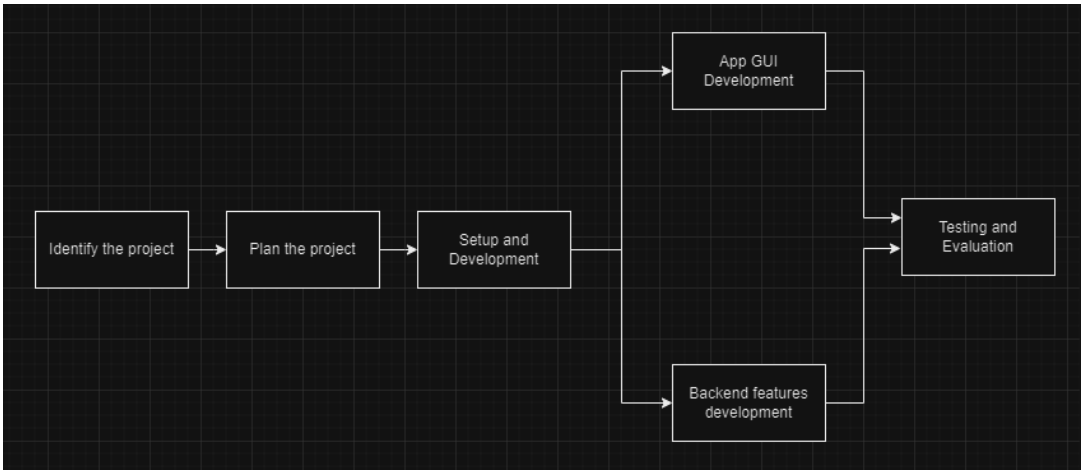


*Figure 1: Planned process of development*

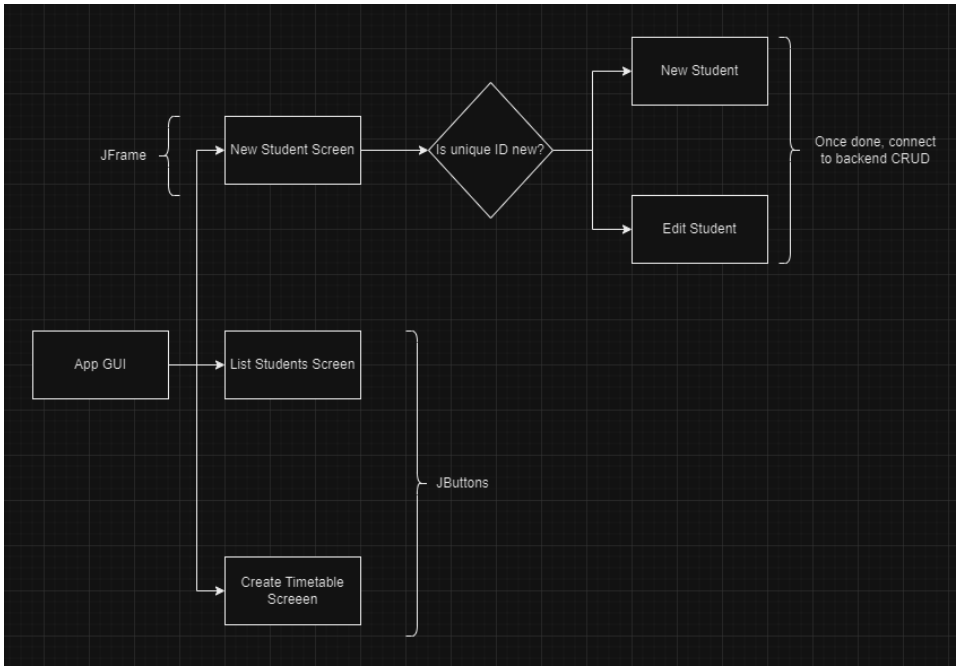


*Figure 2: Planned development of the frontend*

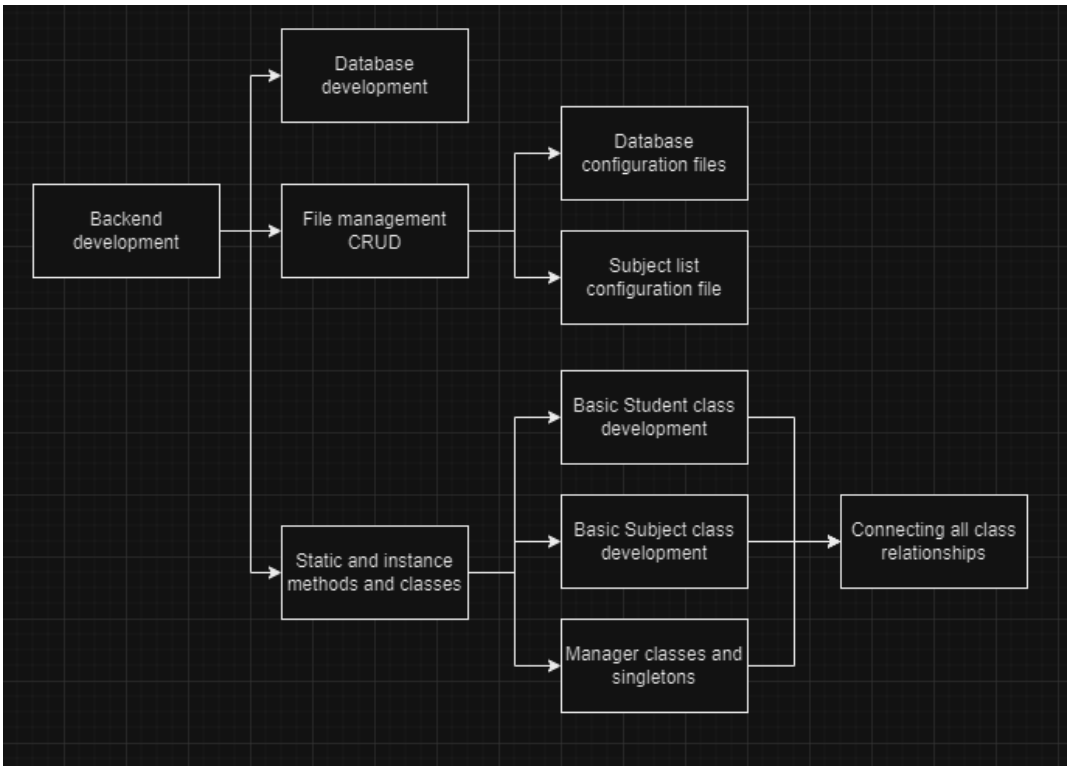


*Figure 3: Backend features development plan*

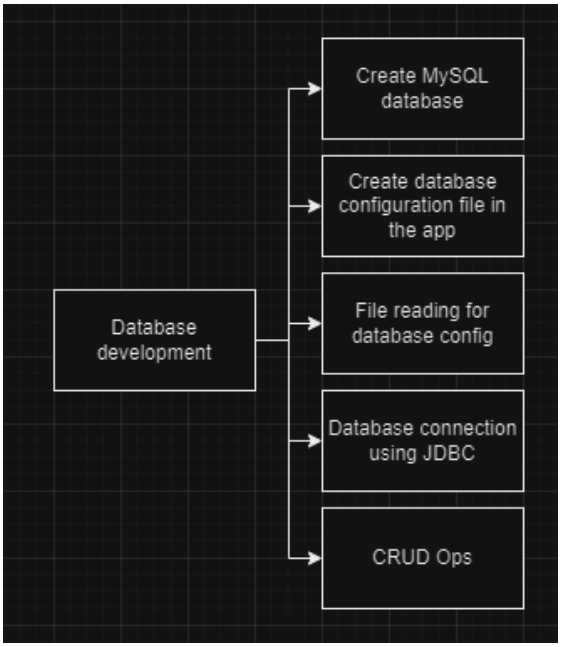


*Figure 4: Decomposition process for database development*

***The Plan***

| Task | Estimated Hours | Timeframe |
|---|---|---|
| Project identification and specification | 1 hour for finding important real-life issue/problem, 1 hour for advising ideal computational solution | 2 days |
| Project planning and design | 1 hour for researching languages, 1 hour for success criteria identification, 2 hours for software sketch, 2 hours for solution overview | 3 days |
| Project setup | 1h30min for setting up the Java project and dependencies, 1h30min to set up the MySQL database | 1 day |
| Backend features development | 1 hour to set up necessary classes and student information blueprints, 2 hours to set up the necessary file configuration settings for offered DP courses, 3 hours to set up all the necessary database read-write operation methods | 2 weeks |
| Frontend features development | 2 hours to construct the GUI, 1 hour to make an option to add new students and information via the GUI, 4 hours to create a balanced timetable algorithm | 2 weeks |
| Testing, fixing and evaluation | 1 hour to personally test the app, 2 hours to fix all noticed issues<br>3 hours for beta testing with any interested stakeholders, 3 hours to improve user experience and fix all bugs | 2 weeks |

# Criterion C – System Overview

## ***System Model***

All diagrams have been created using the draw.io web application.

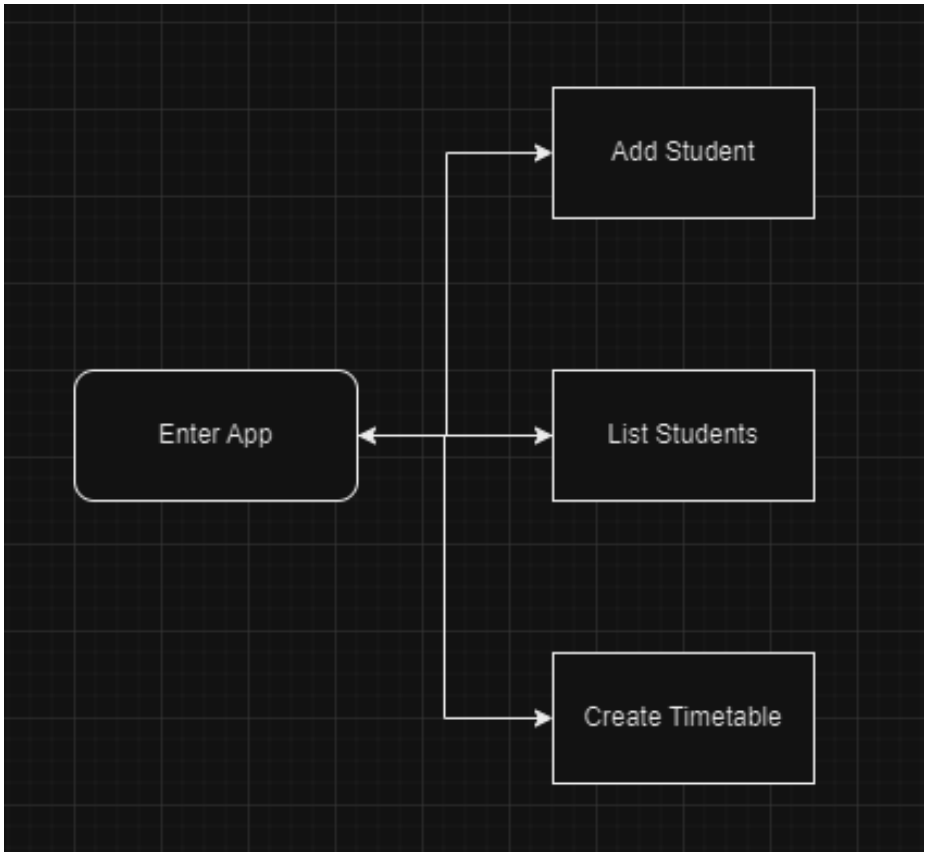


*Image 1: Use cases when app is opened*

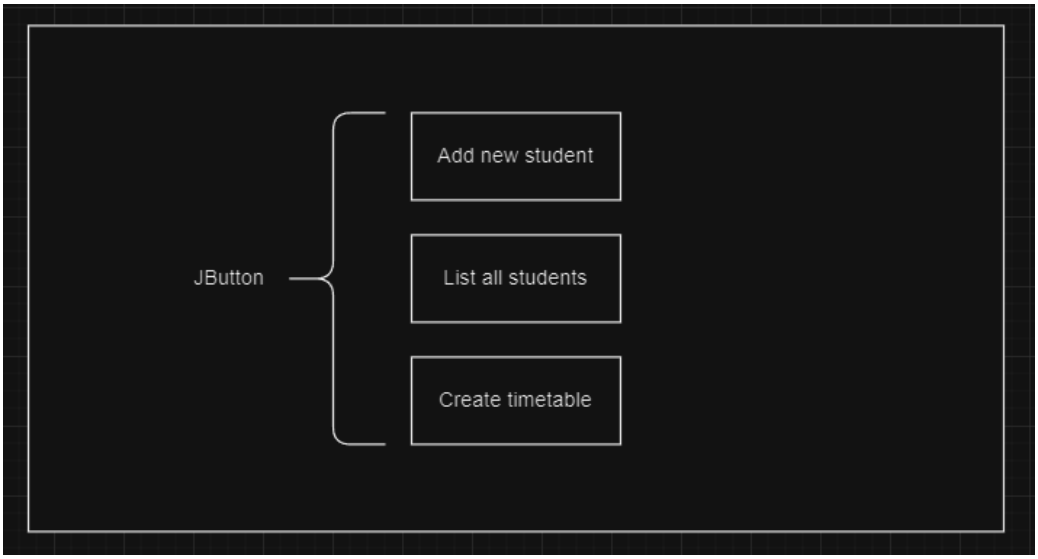


*Image 2: Initial main menu screen*

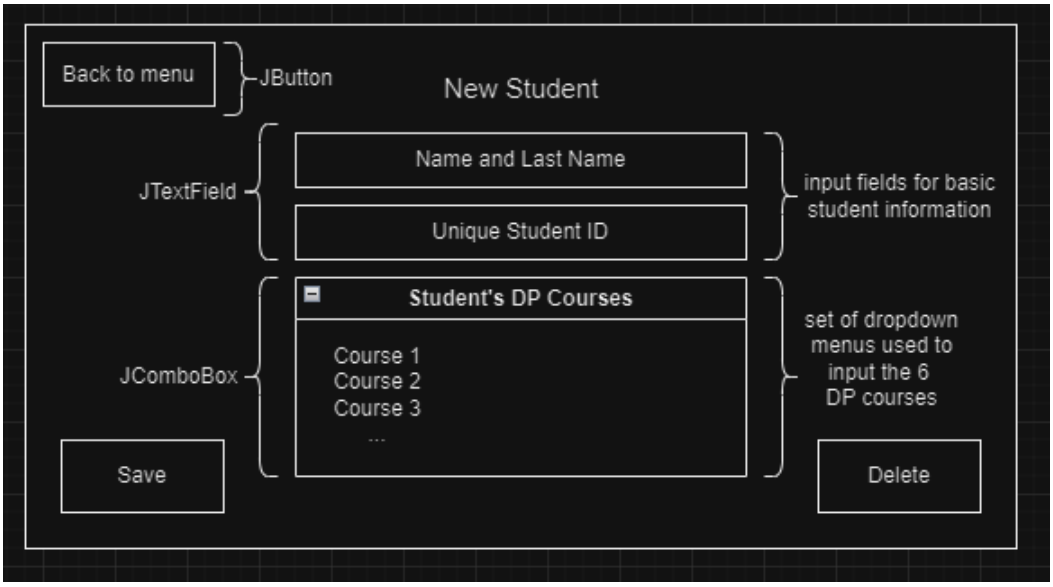


*Image 3: New student creation menu*

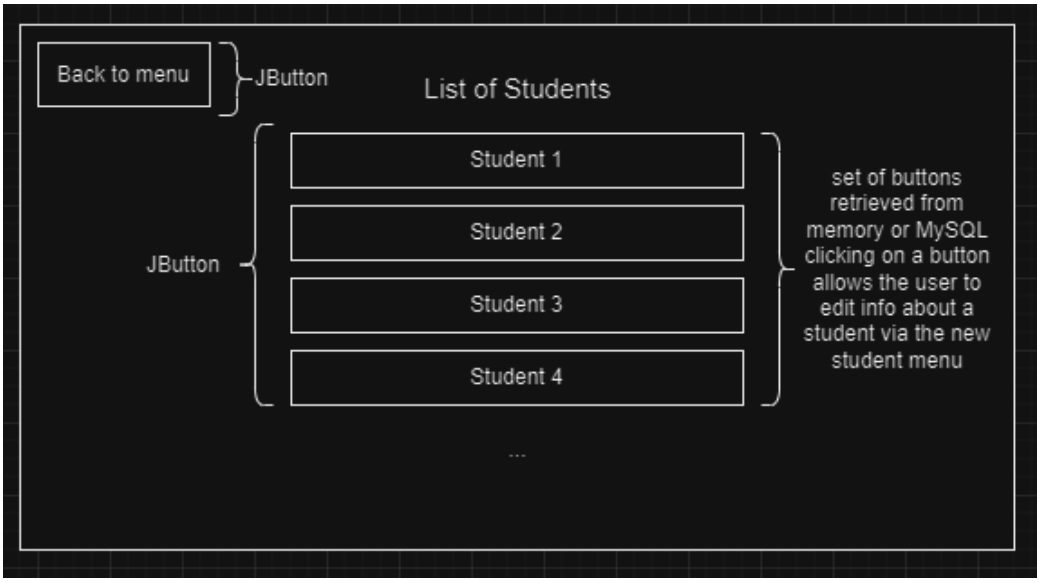


*Image 4: List of students menu*

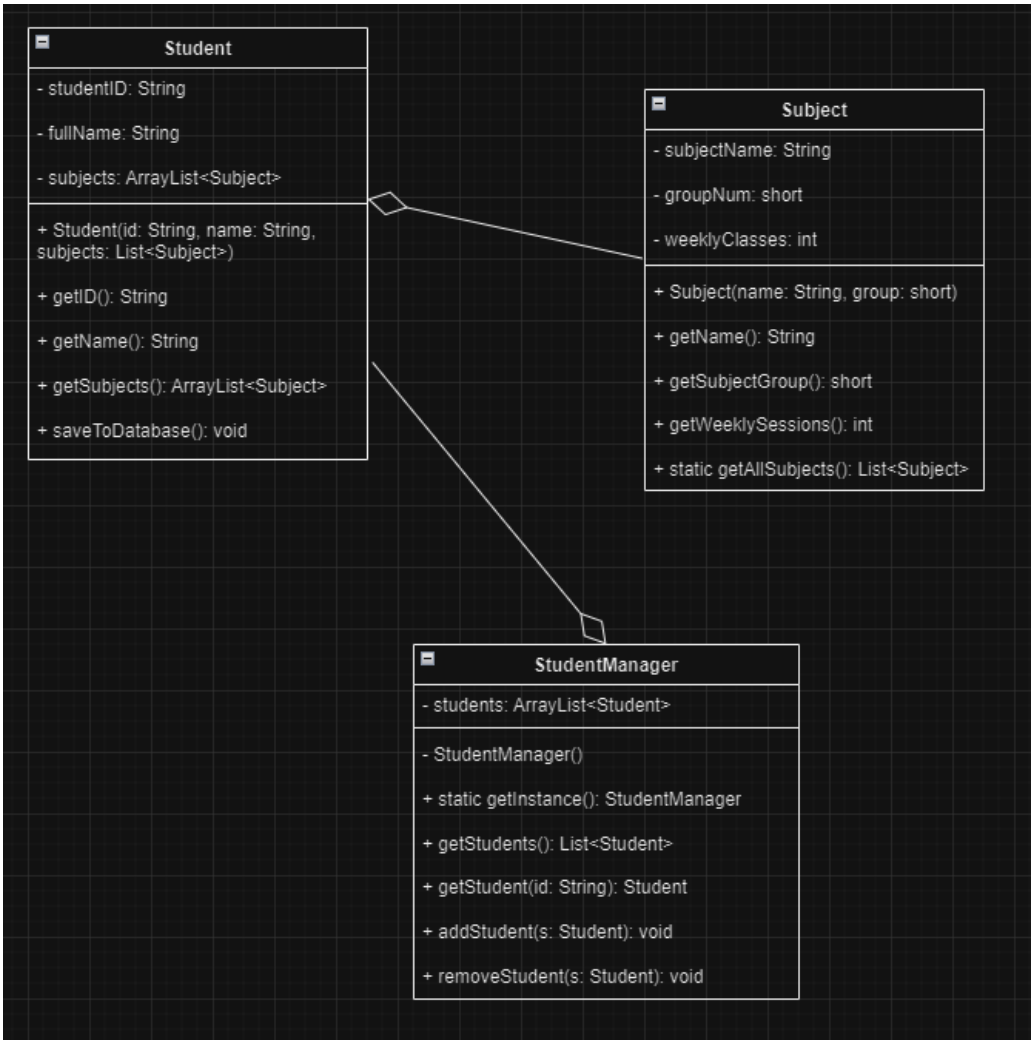


*Image 5: UML Class Diagram*

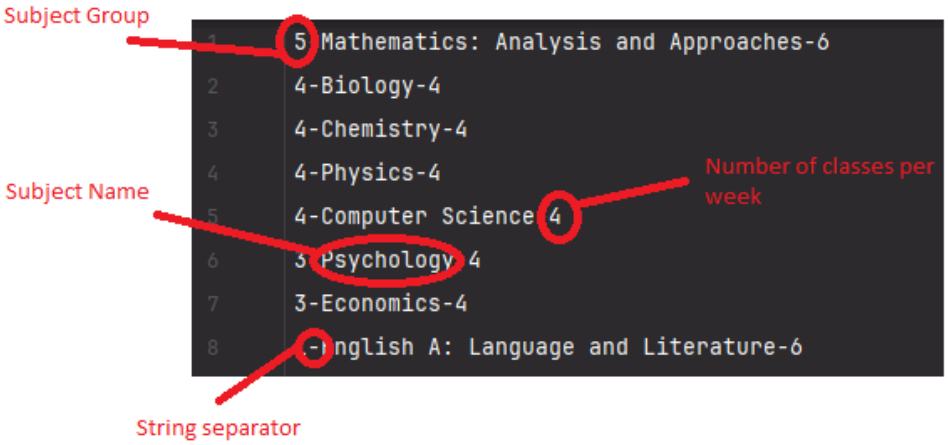


*Image 6: Subject configuration file*

| studentID | fullName | subjects |
|---|---|---|
| xydj12 | Peter Peterson | Biology,Chemistry,Psychol |
| sadg28 | Jack Jackson | History,Geography,Chemis |

*Image 7: Database table structure*

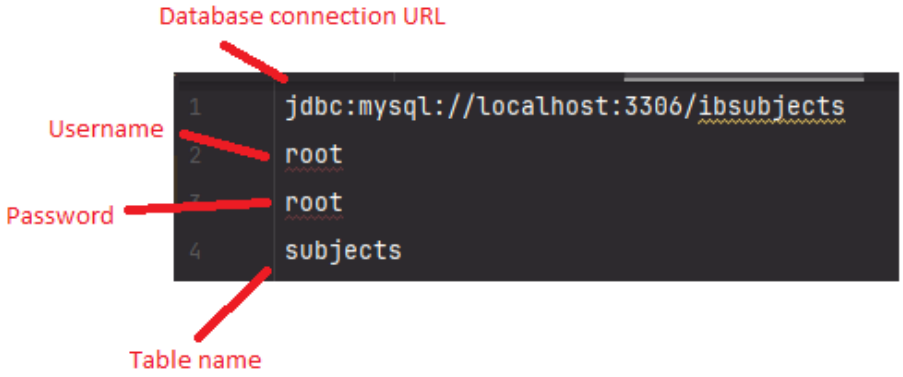


*Image 8: Database configuration file*

## ***Algorithms in pseudocode***

Store all students from memory into the database:

```
function saveToDB():
    students = subjectManager.getInstance().getStudents();
    conn = database.getConnection();
    sql = """
        INSERT INTO students (id, name, subjects) VALUES (?, ?, ?)
        ON DUPLICATE KEY UPDATE name = VALUES(name), grade =
VALUES(grade), subjects = VALUES(subjects)
    """
    statement = conn.prepareStatement(sql)
    try:
        for student in students:
            statement.setString(1, student. getID())
            statement.setString(2, student.getName())
            statement.setString(3, student.getSubjects())
            statement.executeUpdate()
    catch exception:
        output(exception.stacktrace)
    statement.close()
    conn.close()
```

Retrieve all students from the database and store them in memory:

```
function loadStudentsFromDB():
  students = new ArrayList<Student>()
  conn = database.getConnection()
  sql = "SELECT id, name, subjects FROM students"
  statement = conn.prepareStatement(sql)
  try:
    result = statement.executeQuery()
    while result.next():
      studentID = result.getString(1)
      name = result.getString(2)
      subjects = result.getString(3)
      student = Student(studentID, name, subjects)
      studentManager.addStudent(student)
  catch exception:
    output(exception.stacktrace)
  statement.close()
  conn.close()
  return students
```

Method to retrieve how many students take a course:

```
function getSubjectPopularity(subject, studentManager):
  students = studentManager.getStudents()
  count = 0
  for student in students:
    if subject in student.getSubjects():
      count += 1
  return count
```

Method to check whether two subjects share common students:

```
function sharedStudents(subject1, subject2, studentManager):
  students = studentManager.getStudents()
  for student in students:
    if subject1 in student.getSubjects():
          if subject2 in student.getSubjects():
               return true
  return false
```

Method to create timetable:

```
function buildTimetable(studentManager):
  timetable = new List<>()
  currentDayClasses = 0, currentDay = 1
  weeklyClasses = sum (subject.getWeeklyClass())
  maxWeeklyClasses = weeklyClasses / 5
  subjects = sort Subject.getAllSubjects() by getSubjectPopularity
  for subject in subjects:
    if currenyDayClasses >= maxWeeklyClasses:
      currentDayClasses = 0
      currentDay += 1
    for i in subject.getWeeklyClasses():
      timetable[currentDay - 1].add(subject)
  return timetable
```

## ***Test Plan***

| Success criterion | Test | Expected result |
|---|---|---|
| The app will allow the user to input basic data about a student. | Input name and unique IDs for multiple students.<br><br>Input name and matching IDs. | The app successfully registers the student.<br><br>The app will not allow matching user IDs. |
| The app will allow a user to state which 6 IBDP courses a student takes. | Input 6 or more offered IBDP courses for a student.<br><br>Input less than 6 offered courses, or attempt to add courses not offered, or attempt to input one course more than once. | The app successfully registers the student's courses.<br><br>The app will not allow an IBDP student to have fewer than 6 courses, and will only display courses that have been configured as offered, and one course can only be selected once. |
| The app will persistently store all data about a student, including courses. | Add a multitude of students, close and reopen the application multiple times. | The app will display the students who have already been registered in the previous app sessions. |
| The app will be able to automatically create a timetable of classes based on given data.<br><br>The timetable will not feature clashes of two classes unless no students take both courses scheduled in that period.<br><br>Classes taken by fewer students will be put at the front of the schedule so that most students and teachers do not have to arrive at school earlier than necessary. | After registering a large set of students, create a timetable.<br><br>Repeat the process with the same student database.<br><br>Change the student database. | The timetable features all of the subjects which are offered by the school. Subjects that are taken by fewer students are at the front of the schedule. There are no class interlaps in subjects.<br><br>With the same dataset, the app can produce a different timetable, but it will always stay balanced and will respect the success criteria.<br><br>The app will devise a new, balanced timetable. |
| The app user will be able to input which courses are offered by the school using a configuration file, as well as how many times classes in a week that subject features. | Input a set of subjects offered in the configuration file with the proper formatting.<br><br>Use invalid defined string file formatting or an invalid file type. | All of the subjects offered by the school are present in the app and can be selected.<br><br>The app informs the user that the configuration file is corrupt upon opening. |

# Criterion D – Development

## ***Techniques Used***

1. File reading

```
public static List<Subject> getAllSubjects() {
    File file = new File( pathname: "subjects.txt");
    ArrayList<Subject> subjects = new ArrayList<>();
    try {
        for (String line : Files.readAllLines(file.toPath())) {
            String[] parts = line.split( regex: "-");
            int groupNum = Integer.parseInt(parts[0]);
            int weeklyClasses = Integer.parseInt(parts[2]);
            Subject subject = new Subject(parts[1], (short) groupNum, weeklyClasses);
            subjects.add(subject);
        }
    } catch (Exception e) {
        throw new RuntimeException("Error reading subjects file");
    }
    return subjects;
}
```

The above code reads a file in the working directory called 'subjects.txt', which aims to obtain all of the subjects offered by the IBWS from the specified file. Initially, a new File object is instantiated along with a subjects ArrayList to store all of the loaded subjects. Inside of the try statement, a for-each loop is used to loop over all of the lines in the file, each line being split by a string separator '-', mentioned before in this documentation. The $0^{th}$ ($1^{st}$) and $2^{nd}$ ($3^{rd}$) element of the newly obtained array are parsed as integers, whilst the $1^{st}$ ($2^{nd}$, subject name) element is used directly in the constructor of the new subject object, later added in the subjects array.

The reason why I used this approach is because it would be the easiest for a potential IBWS to understand, manage and edit, and the approach in the method is straightforward and the most legible way to read a text file. Evaluating this approach, the advantage of the method is that the file is easy to configure (Success Criterion 7) and file reading and parsing are inside of necessary exception handling techniques, which will catch any exceptions possibly produced by the code. However, the limitations of this method is that the catch statement will run the same code regardless of the exception produced, and will throw a RuntimeException without much feedback, as well as the fact that instead of reading the file upon every call to getAllSubjects(), the results could have been cached in memory to increase performance. Also, the code never checks whether the 'subjects.txt' exists, which could have been fixed by introducing a file.exists() condition.

The testing strategy would be well-employed in this scenario in regards to testing the subjects list file, since that would also employ the testing of the above algorithm, meaning that as long as the testing plan is followed for normal, abnormal and extreme data, the algorithm will be well-tested.

2. Multithreading

```
new Thread(() -> {
    try {
        Connection connection = SQL.getInstance().getConnection();
        String table = SQL.getInstance().getTable();
        PreparedStatement statement = connection.prepareStatement( sql: "INSERT INTO " + table + " (studentID, fullName, subjects) VALUES (?, ?, ?)");
        statement.setString( parameterIndex: 1, studentID.getText());
        statement.setString( parameterIndex: 2, fullName.getText());
        statement.setString( parameterIndex: 3, parseSubjects(studentSubjects));
        statement.executeUpdate();
    } catch (Exception ex) {
        throw new RuntimeException("Error saving student to database");
    }
}).start();
```

```
private String parseSubjects(ArrayList<Subject> subjects) {
    StringBuilder sb = new StringBuilder();
    for (Subject s : subjects) {
        sb.append(s.getSubjectName()).append(", ");
    }
    return sb.toString();
}
```

In order to not damage the app's performance or block the main thread of the app, whenever an SQL statement is created and executed, a new thread is made that performs said database update in order to run tasks concurrently. A new thread object is created, and a lamba function with no parameters (which is used instead of instantiating a Runnable) is used as the parameter to the Thread constructor. A PreparedStatement is initialized inside of this lamba function in order to insert a new student into the table that is referenced inside of the SQL singleton. The values are represented by '?' in the statement, and are later replaced by the library using the statement.setString(index, value) method. The parseSubjects(list) method will use a StringBuilder to create a  After this, the statement is executed in the database.

I used this approach because it can make the app run SQL updates asynchronously, which can significantly improve performance and reduce the load on the main thread of the app, and the SQL database will persistently store all of the student data even when the app is closed (Success Criterion 3). Concurent running, performance and multithreading are also the advantages of this approach, and so is exception handling. However, there are also issues that can arise with this approach, such as debugging difficulties.

The testing strategy is partially effective in the above scenario, since it can be well-used to test the functionality of the database CRUD operations, however, it does not explicitly reference multithreading itself.

3. Singletons

```
class StudentManager {
    5 usages
    private final ArrayList<Student> students;
    3 usages
    private static StudentManager instance;

    1 usage  new *
    private StudentManager() {
        students = new ArrayList<>();
    }

    1 usage  new *
    public static StudentManager getInstance() {
        if (instance == null) {
            instance = new StudentManager();
        }
        return instance;
    }

    no usages  new *
    public List<Student> getStudents() {
        return Collections.unmodifiableList(students);
    }
```

Since I wanted to encapsulate the list that stores all of the students, I decided to create a StudentManager class which will be a singleton. A singleton is a class that can only be instantiated once (only has one object). The way this is achieved is by privatizing the class' constructor and using a static variable containing the single instance that will be instantiated only the first time the static getInstance() accessor method is called, and will only return the already instantiated variable upon every subsequent call. The getStudents() method will return an immutable copy of the students list, protecting its contents from improper access.

The advantage of this approach and the reason why I've used it is because it encapsulates the students list and doesn't allow any class apart from StudentManager to modify it. The limitation of this approach is that in many scenarios, singletons may be unnecessary to use instead of just regular accessor and mutator methods.

4. SQL database

```
class SQL {

    3 usages
    private static SQL instance;
    3 usages
    private final Connection connection;
    3 usages
    private final String table;

    1 usage  new *
    private SQL() {
        File file = new File( pathname: "database.txt");
        if (!file.exists()) {
            try {
                file.createNewFile();
            } catch (Exception e) {
                throw new RuntimeException("Error creating database file");
            }
        }
        try {
            List<String> lines = Files.readAllLines(file.toPath());
            String url = lines.get(0);
            String username = lines.get(1);
            String password = lines.get(2);
            table = lines.get(3);
            Class.forName( className: "com.mysql.jdbc.Driver");
            connection = DriverManager.getConnection(url, username, password);
            PreparedStatement statement = connection.prepareStatement( sql: "CREATE TABLE IF NOT EXISTS " + table + " (studentID VARCHAR(255), fullName VARCHAR(
            statement.executeUpdate();
        } catch (Exception e) {
            throw new RuntimeException("Error reading database file");
        }
    }
```

The SQL class, another singleton, is used to connect and keep the connection to the MySQL database, the techniques used in this class were mostly shown in previous parts of this document. The important things to note is that this class, at construction, reads the database.txt file and its first 4 lines, representing the database credentials, and then uses the Java MySQL connector library to connect, at which point it will create the specific database table if needed. This singleton is instantiated in the main method of the program, before the app's GUI is opened.

The advantages of this approach are exception handling and checking for file existance, the ability to change database credentials and to access the singleton safely in order to create further PreparedStatements. It is important to note that the limitation of this is that the app will not work without the database with a constant connection, without a stable internet connection, as well as proper file configuration setup.

The testing plan can be very effective to test all SQL-related algorithms, including the above, as testing the persistency-related success criteria will also employ the testing of these algorithms, meaning that by following the testing strategy for all inputs, these algorithms will also be well-tested and evaluated.

5. Timetable creation

```
private List<String> buildTimetable() {
    List<StringBuilder> timetable = new ArrayList<>();
    for (int i = 0; i < 5; i++) {
        timetable.add(new StringBuilder());
    }
    AtomicInteger currentDay = new AtomicInteger( initialValue: 1);
    AtomicInteger currentDayClasses = new AtomicInteger( initialValue: 0);
    AtomicInteger maxClassesPerDay = new AtomicInteger( initialValue: 0);
    for (Subject subject : Subject.getAllSubjects()) {
        maxClassesPerDay.getAndAdd(subject.getWeeklyClasses());
    }
    maxClassesPerDay.set(maxClassesPerDay.get() / 5);
    HashMap<Subject, Integer> subjects = new HashMap<>();
    for (Subject subject : Subject.getAllSubjects()) {
        subjects.put(subject, getSubjectPopularity(subject));
    }
    subjects.entrySet().stream().sorted(Comparator.comparingInt(Map.Entry::getValue)).forEach(s -> {
        if (currentDayClasses.get() >= maxClassesPerDay.get()) {
            currentDayClasses.set(0);
            currentDay.getAndIncrement();
        }
        for (int i = 0; i < s.getKey().getWeeklyClasses(); i++) {
            timetable.get(currentDay.get() - 1).append(s.getKey().getSubjectName()).append("; ");
        }
        currentDayClasses.set(currentDayClasses.get() + s.getKey().getWeeklyClasses());
    });
    List<String> result = new ArrayList<>();
    for (StringBuilder sb : timetable) {
        result.add(sb.toString());
    }
    return result;
}
```

The above code is responsible for building the timetable. The method first initializes a timetable list of StringBuilders, which will store which classes are on which day, which is why 5 are added via a for loop. After that, three AtomicIntegers are created instead of regular integers so that they can be used inside of a lamba functions, the total number of classes per week and per day are calculated, and after that, all subjects are stored in a map where the key is the subject, and the value is the subject popularity, after which they are sorted using a stream by value, and looped in ascending order (in order to satisfy Success Criteria 5 and 6), where in the loop, all of the weekly classes are added to the current day, unless the max number of daily classes has been reached. After that, the resulting list is returned as a list of Strings.

The advantage and reason for using this approach is that it creates a balanced timetable for all 5 working days of the week and sorts the subjects by popularity. However, a notable disatvantage of this algorithm is the fact that it features a lot of looping, which can be damaging to performance if a school offers a wide range of subjects and has many students.

Once again, by testing the timetable creation success criteria, we are also testing the above algorithm and can evaluate its functionality and effectiveness, meaning that the testing strategy in this scenario is effective and employed.

# Criterion E – Evaluation

## ***Evaluation of the Success Criteria***

1. **The app will allow the user to input basic data about a student:** The app allows the user to input, edit, and delete students with all the necessary information. More fields could be added, but it was unnecessary in this scenario.
2. **The app will allow a user to state which 6 IBDP courses a student takes:** The app allows to both state and edit the IBDP courses of every student. There could have been a possibility to allow for extra courses.
3. **The app will persistently store all data about a student, including courses:** With a proper database setup and connection, and with a stable internet connection, the user can modify database credentials and connect to it, and the data is kept persistently in the database as long as it is turned on.
4. **The app will be able to automatically create a timetable of classes based on given data:** Based on which courses are offered by the school and which courses the students take, the app can form a timetable for 5 working days of the week.
5. **The timetable will not feature clashes of two classes:** Every subject and class is separate from one another and do not clash.
6. **Classes taken by fewer students will be put at the front of the schedule so that most students and teachers do not have to arrive at school earlier than necessary:** The classes with the lowest popularity are always put at the start of the week, so that the students and teachers only arrive on days they must. Perhaps a better way to do this could have been to have the lower popularity classes at the start of every day instead of at the start of every week.
7. **The app user will be able to input which courses are offered by the school using a configuration file, as well as how many times classes in a week that subject features:** The app features a file 'subjects.txt' that is run from the working app directory, that allows the school to add, modify or edit subjects and DP courses that are offered by the school, along with stating in which IBDP course group they belong, and how many times that subject should have classes per week.

Overall, all of the success criteria have been mostly or fully met, with some recommendations and room for improvement.

***Justifications for improvement***

The most notable thing that can be improved is the timetable building algorithm. Currently, whilst it does sort the subjects based on the amount of students taking them, it places subjects with low popularity all together at the start of the week, instead of putting them at the start of every day, which could be a more benefitial and wanted solution for some IB schools. The GUI could also be significantly improved in order to have a more visually appealing look, since the current app GUI was aimed for functionality over user experience or overall visual representation.

Another improvement could be the ability to allow schools to assign more than 6 subjects to their students, since, while the students must pick 6 courses in order to fulfill IBDP subject criteria, they can also pick extra courses, which a lot of schools allow. A way to achieve this would be to add an option to add more dropdown menus into the New Student menu. Another possible and benefitial improvement would be to introduce more and enhance current exception handling, as it provides little information on the exceptions that could arrise in SQL connections, file reading, etc. More information should be provided on how to set up the necessary configuration files and databases for user experience, and a good improvement for file management would be to use a format such as JSON or YAML instead of regular text files.