İZMİR EKONOMİ ÜNİVERSİTESİ

Design Document for "Exam Timetable"

Group 4

Ece Naz Gökalp 20210602028

Sine Öykü Yaşar 20230602074

Kaan Bilgili 20210602012

Karya Tanrıkulu 20230602064

Ayşenaz Gelen 20230602027

Ata Kaan Karakuş 20220602045

Index

1 Introduction

The "Exam Scheduler" is a standalone desktop application designed to help the student affairs department in the process of generating a conflict-free exam timetable that can be used during the university's examination period. The main purpose of this system is to automate the process of creating a feasible exam schedule that meets the required constraints of the institution by using data provided by the user. The application is designed to run on Windows, ensuring compatibility with the institution's existing infrastructure.

The scope of the system consists of importing required data, applying scheduling constraints, and producing a visualized output. The user workflow starts with the import of data as CSV files, which includes a list of students, courses, and classrooms. This approach eliminates the need for manual data entry, allowing the user to easily update input data by modifying the CSV files and re-importing them when needed. The final output is a generated schedule that can be viewed either within the application or exported for external use.

To ensure the generated schedule is both feasible and fair to students, the system design is constrained by specific functional requirements defined in the requirements document. The software architecture detailed in this document is built to satisfy the following system requirements:

- **Exams Per Day Limitation (Functional Requirement 8):** The system shall ensure that the number of exams a student has in one day does not exceed two.

- **Consecutive Exam Prevention (Functional Requirement 9):** The system shall generate a schedule where a student does not have two consecutive exams in a day.

- **Capacity Validation (Functional Requirement 10):** The system shall ensure that classroom capacities are not exceeded.

- **Conflict Detection (Functional Requirement 11):** The system must detect and prevent direct time conflicts, ensuring no student is assigned to two exams occurring simultaneously.

The design outlined in this document uses a layered architecture combined with Model-View-Controller (MVC) principles to meet these requirements. It features a core set of domain entities (Students, Courses, Classrooms) wrapped in a scheduling logic engine that adheres to all defined constraints.

In addition to functional requirements, the design also deals with several non-functional requirements that were mentioned in the Requirements Document. A Graphical User Interface (GUI) is designed for the application, which ensures that the program meets modern software standards and user of the application can navigate the system easily. In order to

maintain system stability, the architecture includes error handling to display clear messages about what went wrong rather than simply crashing in the event of unexpected errors or unresolvable scheduling conflicts. Finally, to meet the institution's preexisting standards, all interface elements are presented in English.

2 Software Architecture

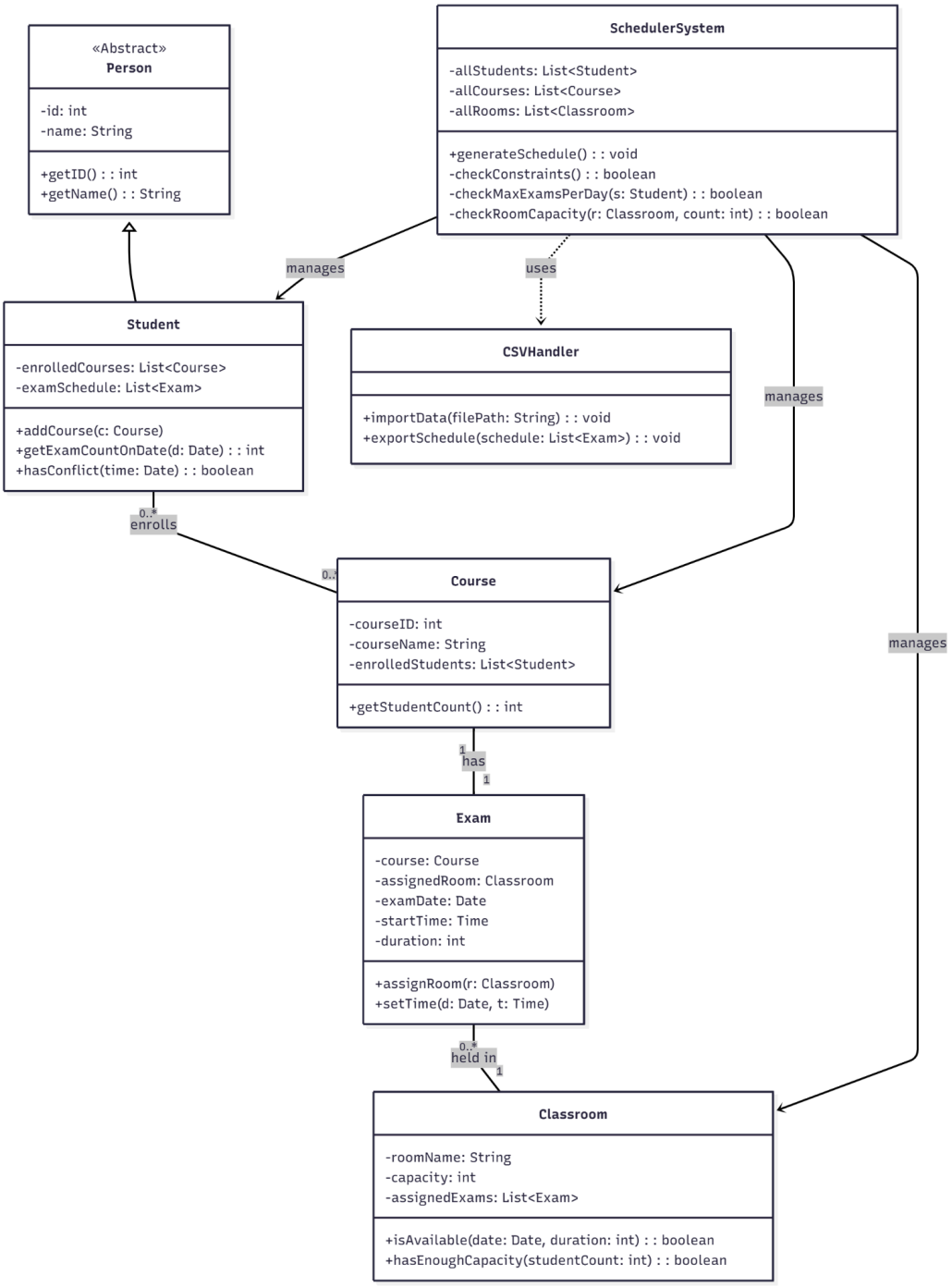
This section defines the high-level structure of the application and explains how its main components collaborate in order to satisfy the functional and non-functional requirements. The system is designed in a way that it can import institutional data, apply scheduling constraints, generate conflict-free exam timetables, and present them to the user through an intuitive interface. This architecture is designed using a layered approach combined with Model–View–Controller (MVC) principles. At the core of the system lie the domain entities—such as students, courses, classrooms, exams, and timeslots *(called core entities or elements)*—which represent the fundamental elements required for generating an exam schedule. Surrounding this domain layer is the scheduling engine responsible for constructing valid timetables while enforcing constraints.

The upper layers provide data management and user interaction capabilities. The data management layer handles CSV import/export and maintains all domain objects, while the presentation layer manages the graphical user interface for configuration, visualization, and system control. Controllers coordinate these layers by receiving user requests, invoking scheduling logic, and updating the interface accordingly.

Overall, this architecture ensures that the system remains modular, testable, and adaptable to changes—such as adding new constraints, modifying the scheduling logic, or extending the GUI—while maintaining a clear separation of concerns.

2.1 Structural Design (Class Models)

The **structural design** of the "Exam Scheduler" system is focused on a set of key classes that represent the core entities required to generate and manage the exam schedules, following the specified requirements. Each class is designed with a clear responsibility, ensuring a modular and extensible system. The structure is meant to address both the functional and non-functional requirements by making the system flexible, scalable, and maintainable



UML of the Classes in Exam Scheduler

2.1.1 SchedulerSystem Class

Definition: The **SchedulerSystem** class serves as the main engine of the application. It encapsulates the core business logic required to generate the exam timetable. It maintains the global state of the application by holding lists of all Student, Course, and Classroom objects loaded into the system. It is responsible for triggering the scheduling algorithm and orchestrating the validation rules.

Relationships: This class acts as a manager. It possesses aggregate associations with **Student, Course, and Classroom** classes, as it manages collections of these objects (e.g., allStudents, allCourses). It has a dependency relationship with the CSVHandler class, utilizing it to import raw data and export the final result. It does not inherit from any other class but serves as the primary entry point for the scheduling logic.

Support for Requirements:

Functional Requirement 8: Ensures that no student has more than two exams on the same day. This is checked by the **checkMaxExamsPerDay()** method, which is called during schedule generation.
Functional Requirement 9: Ensures that no student has consecutive exams. The **checkConstraints()** method enforces this by analyzing the generated schedule.
Functional Requirement 10: Ensures that classroom capacities are not exceeded. This is handled by the **checkRoomCapacity()** method, which checks if a classroom has sufficient seats for the number of students.

```
public class SchedulerSystem{ public List<Student> allStudents;

   public List<Course> allCourses;

   public List<Classroom> allRooms;

 public bool generateSchedule() {

      // Logic to generate the exam schedule

   } public bool checkConstraints() { // Logic to ensure no conflicts
exist

   } public bool checkMaxExamsPerDay(Student student) {  // Logic
to ensure each student doesn't have more than 2 exams per day

   }
```

```
public bool checkRoomCapacity(Classroom room, int count){
// Logic to ensure the classroom has enough capacity
                                    }
                                    }
```

2.1.2 Student Class

Definition: The **Student** class represents the individual learner within the system. Inheriting from the abstract **Person** class, it stores essential identity information alongside academic data, such as the list of enrolled courses and the personal exam schedule.
Relationships: Each student is linked to a list of **Course** objects they are enrolled in.
Each student has an associated **ExamSchedule** for their exam timetable.
Support for Requirements: The structure of the **Student** class is essential for personalizing the schedule validation process.The class includes a method called **getExamCountOnDate**, which allows the system to query exactly how many exams a specific student has on a given day. This data is the foundation for enforcing **Functional Requirement 8**.

Additionally, the **hasConflict** method checks the student's personal timeline to ensure that no two exams overlap, which directly satisfies **Functional Requirement 11** regarding the detection and prevention of scheduling conflicts

```
from datetime import date
class Student:
    def __init__(self, student_id, name):
        self.student_id = student_id
        self.name = name
        self.enrolled_courses = []
        self.exam_schedule = []
def get_exam_count_on_date(self, target_date:
date) -> int:
 count = 0
  for exam in self.exam_schedule:
   if exam.date == target_date:
                count += 1
        return count
   def has_conflict(self, time) -> bool:
        # Logic to check for overlapping times
        return False
```

2.1.3 Course Class

Definition: The **Course** class defines an academic course and the students enrolled in it. It contains details like course ID, course name, and the list of students in the course.

Responsibilities:Tracks students enrolled in the course (enrolledStudents).
Provides the number of students in the course (getStudentCount()).

Support for Requirements: This class provides the critical data needed for resource allocation. By maintaining a list of enrolled students, the **Course** class provides a getStudentCount method. The SchedulerSystem uses this exact count to find a suitable classroom, ensuring that **Functional Requirement 10** is met by preventing the assignment of a room that is too small for the class size.

```
class Course:
 def __init__(self, course_id,
course_name):
self.course_id = course_id
self.course_name = course_name
self.enrolled_students = []
def get_student_count(self) -> int:
return len(self.enrolled_students)
```

2.1.4 Exam Class

Definition and Responsibilities: The **Exam** class represents the final output entity of the scheduling process. It creates a concrete link between a **Course**, a specific **Classroom**, and a designated **time slot** (date and duration).

Support for Requirements: The Exam class is the container for the generated schedule data.By storing the **assignedRoom, date, and startTime**, this class holds the information necessary for the user to review the schedule visually, satisfying **Functional Requirement 12**. Furthermore, because this class aggregates all logistical details into one object, it allows the system to easily export the final timetable, supporting **Functional Requirement 13**

```
class Exam:
def __init__(self, course):
self.course = course
self.assigned_room = None
self.exam_date = None
self.start_time = None

 def assign_room(self, classroom):
 self.assigned_room = classroom
 def set_time(self, date, time):
 self.exam_date = date
 self.start_time = time
```

2.1.5 Classroom Class

Definition and Responsibilities: The Classroom class represents the physical infrastructure available for exams. It defines static properties, such as the room name and total seating capacity, and dynamic properties, such as availability during specific time slots.

Support for Requirements: This class acts as a gatekeeper for physical resources.The **hasEnoughCapacity** method performs a direct integer comparison between the room's limit and the student count, which is the primary mechanism for satisfying **Functional Requirement 10**. It also tracks availability to ensure that the facility is used efficiently without double-booking.

```
class Classroom:
 def __init__(self, room_name, capacity):
 self.room_name = room_name
self.capacity = capacity
 def has_enough_capacity(self, student_count: int) -> bool:
return self.capacity >= student_count
def is_available(self, date, duration) -> bool:
return True
```

2.1.6 CSVHandler Class

Definition and Responsibilities: The **CSVHandler** class is a specialized utility designed to manage data persistence. It handles the technical task of reading external CSV files to populate the system's objects and writing the generated schedule back to a file format.

Support for Requirements: This class is the interface between the system and the external environment. It implements the **importData** method to parse incoming files, which directly satisfies **Functional Requirement 1** regarding the importation of necessary data and **Functional Requirement 6** regarding parsing correctness.

Additionally, it provides the exportSchedule method, enabling the user to save and share the final results, thus fulfilling **Functional Requirement 5** and **Functional Requirement 13**.

```
import csv
class CSVHandler:
def import_data(self, file_path: str):
# Satisfies Requirement 1 & 6
 # Logic to read and parse CSV files
def export_schedule(self, schedule: list):
# Satisfies Requirement 5 & 13
 # Logic to write the schedule to a CSV file
```

2.2 Behavioural Design

The core objects and the scheduler engine must collaborate at runtime in such a way that each course is assigned a valid exam slot and room while ensuring that all student, time, and capacity constraints are satisfied and that the resulting schedule remains consistent across all interacting objects. The term "Runtime Scenario" is introduced to summarize the behaviour of the program during different parts of its execution while satisfying the constraints and ensuring consistency.

The key runtime scenarios are:

1. Importing institutional data from CSV files
2. Generating an exam schedule
3. Checking and enforcing constraints during scheduling
4. Exporting the generated schedule

Each scenario below explains the interaction between SchedulerSystem, CSVHandler, Student, Course, Exam, and Classroom.

2.2.1 Responsibilities of Each Class and How They Collaborate

The behavioral responsibilities of each class are as followed:

SchedulerSystem Responsibilities:

- Orchestrating high-level operations: **generateSchedule, checkConstraints, checkMaxExamsPerDay, checkRoomCapacity.**
- Maintaining the collections allStudents, allCourses, allRooms used during scheduling.

CSVHandler Responsibilities:

- Handling all file I/O: **importData** to populate the system, and **exportSchedule** to persist the final list of exams.

Student Responsibilities:

- Maintaining its list of enrolled courses and scheduled exams.
- Answers questions about exam counts and conflicts on a particular date via **getExamCountOnDate** and **hasConflict.**

Course Responsibilities:

- Maintaining its enrolled students and exposing **getStudentCount** for capacity checks.

Exam Responsibilities:

- Representing a concrete exam instance for a course, with assigned room and time.
- Allowing the scheduler to assign room and time via **assignRoom** and **setTime.**

Classroom Responsibilities:

- Keeping track of exams assigned to it.
- Answering availability and capacity queries via **isAvailable** and **hasEnoughCapacity**.

All subsequent scenarios are built on these responsibilities.

2.2.2 Importing Data from CSV Files

The program's first goal is to load students, courses, enrollments and classrooms into the system upon users request. When the user selects the CSV files in the GUI, an *import* action is triggered. CSVHandler invokes the **importdata(filePath: String)** method. Inside CSVHandler.importdata:

- The CSV file(s) are parsed line by line.
- For each student record: A **Student instance** (subclass of Person) is created if it does not already exist. The student is added to **SchedulerSystem.allStudents**.
- For each course record: A **Course instance** is created (or retrieved if existing). The course is added to **SchedulerSystem.allCourses.**
- For each enrollment relation (between student–course): **Student.addCourse(course)** is called.The course also adds the student to **enrolledStudents** (e.g. internally in addCourse or in CSVHandler).
- For each classroom record: A **Classroom instance** is created and added to **SchedulerSystem.allRooms**.

2.2.3 Generating the Exam Schedule

The second goal of the program is to assign each course an exam with a date, starting time and classroom, while respecting constraints. When the user triggers **GenerateSchedule** in the GUI, **SchedulerSystem.generateSchedule()** is called. SchedulerSystem prepares an initially empty collection of Exam objects (e.g. a local list or part of the system state). For each Course in allCourses:

1. A new Exam object is created that is associated with that course.
2. SchedulerSystem selects a candidate date and start time according to the configured exam period and slots.
3. SchedulerSystem selects a candidate Classroom from allRooms.
4. Before committing the assignment, SchedulerSystem calls:

    a. **checkMaxExamsPerDay(student)** for each enrolled student.
    b. **checkRoomCapacity(candidateRoom, course.getStudentCount())**.
    c. **Classroom.isAvailable(candidateDate, duration)**.
    d. For each enrolled student, **Student.hasConflict(candidateDate)** (or the time-based equivalent).

5. If all checks pass:

    a. **Exam.setTime(date, startTime)** is called.
    b. **Exam.assignRoom(classroom)** is called.
    c. The exam is added to both each enrolled student's examSchedule list and the classroom's assignedExams list.

6. If a conflict is detected:

    a. SchedulerSystem tries another time or room and repeats the checks.
    b. If no combination works, the system may mark the course as unschedulable and report it to the user.

As a result each successfully scheduled course will have an Exam object that includes its examDate, startTime, duration and assignedRoom. Students examSchedules will be populated as well as the classrooms assignedExams lists.

2.2.4 Constraint Evaluation During Scheduling

This section describes how SchedulerSystem.checkConstraints, checkMaxExamsPerDay and checkRoomCapacity interact with other objects. During the scheduling process, the system continuously evaluates multiple constraints to ensure that each assigned exam is valid and

non-conflicting. These checks are performed by the SchedulerSystem class in collaboration with Student, Course, and Class objects. The constraint assessment workload is as follows:

-For each proposed exam time slot, the scheduler queries every registered student to verify the following:

a)The scheduler uses the **getExamCountOnDate** function to ensure that no student has exceeded their daily allowed number of exams.

b)The scheduler calls the **hasConflict** function to check if the new exam conflicts with or is too close to another exam in the student's personal schedule.

-Before assigning a room, the scheduler confirms the following:

a)The room must have enough seating capacity for the number of registered students. This is performed using **hasEnoughCapacity**.

b)The scheduler calls the **isAvailable(date, duration)** function to ensure the room is not scheduled for another exam.

-SchedulerSystem.checkConstraints() provides a unified way to combine the above checks. If any constraint fails, the system rejects the suggested time slot or class and tries an alternative combination.

2.2.5 Exporting the Schedule

When the planning process is complete, the system provides an export function for the generated exam schedule. The export process is performed by the **CSVHandler** class, which converts all planned exam data into a structured output file. The export process includes the following:

1. Collecting Exam Information

   The system compiles all Exam objects created during planning. These include:

   -Course name and ID

   -Assigned class

   -Exam date and start time

   -Exam duration

   -Additional planning metadata required for reporting

2. Formatting the Output:

CSVHandler.exportSchedule converts internal planning into a structured CSV format that is easy to read, store, and share. This ensures compatibility with external tools commonly used by student affairs staff.

3. File Creation and User Interaction

   When the user selects the “Export Schedule” option in the GUI, the system initiates the export function. The resulting CSV file represents the final, verified schedule and can be imported into institutional systems, emailed to instructors, or displayed on university platforms.

This export mechanism ensures that the output schedule is not only valid and conflict-free, but also accessible, understandable, and easy to distribute.

3 Graphical User Interface (GUI)

The system is mandated to provide a user-friendly and highly accessible Graphical User Interface (GUI) developed specifically for the Windows operating environment. A crucial requirement is high usability, ensuring that the software is efficient, effective, and satisfactory for generating examination timetables. All interface elements, operational texts, dialog box content, and integrated help messages must be exclusively in English to maintain language consistency and clarity for the target user base.

3.1 Menu Bar

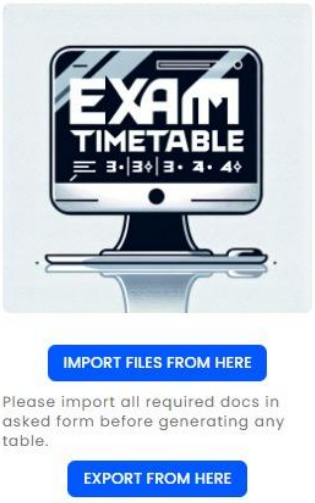
The Menu Bar is the primary navigational component, providing systematic and immediate access to the application's core functionalities.

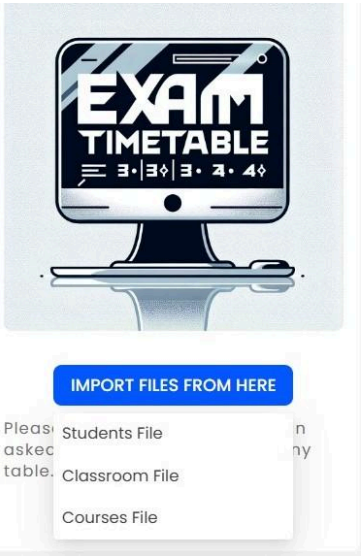
Import Data: Initiates the **ReadAndParseCSV** function to securely load the necessary input data. This action opens a standard file selection dialog, prompting the user to select required files.

Note: This function validates the integrity and format of the selected CSV files prior to loading to prevent runtime errors.

Export Schedule: Initiates the **ExportSchedule** function to save the final, generated timetable. This operation should offer the user a choice between exporting to a CSV or an Excel (.xlsx) file format.

Exit: Closes the application completely. Before exiting, the system must check for any unsaved or pending timetable generation results and prompt the user to confirm the closure if data loss is probable.





3.2 Main Screen Layout and Workflow

The central screen focuses on a clear, three-step workflow: Import, Generate, and Export.

3.2.1 Input Area & Prompts:

- **Headline:** The prominent "EXAM TIMETABLE" display reinforces the application's purpose.
- **Instructional Text:** Clear instruction: "Please import all required docs in asked form before generating any table." This text guides the user to complete the data prerequisite.
- **Expected Files Messages:** This section provides transparency regarding the necessary data sources.
  - **Required Files:** The user must load the following files for successful operation:
    - Students File
    - Classroom File
    - Courses File
  - The GUI must visually indicate which files have been successfully loaded and which are still missing.

3.2.2 Action Buttons:

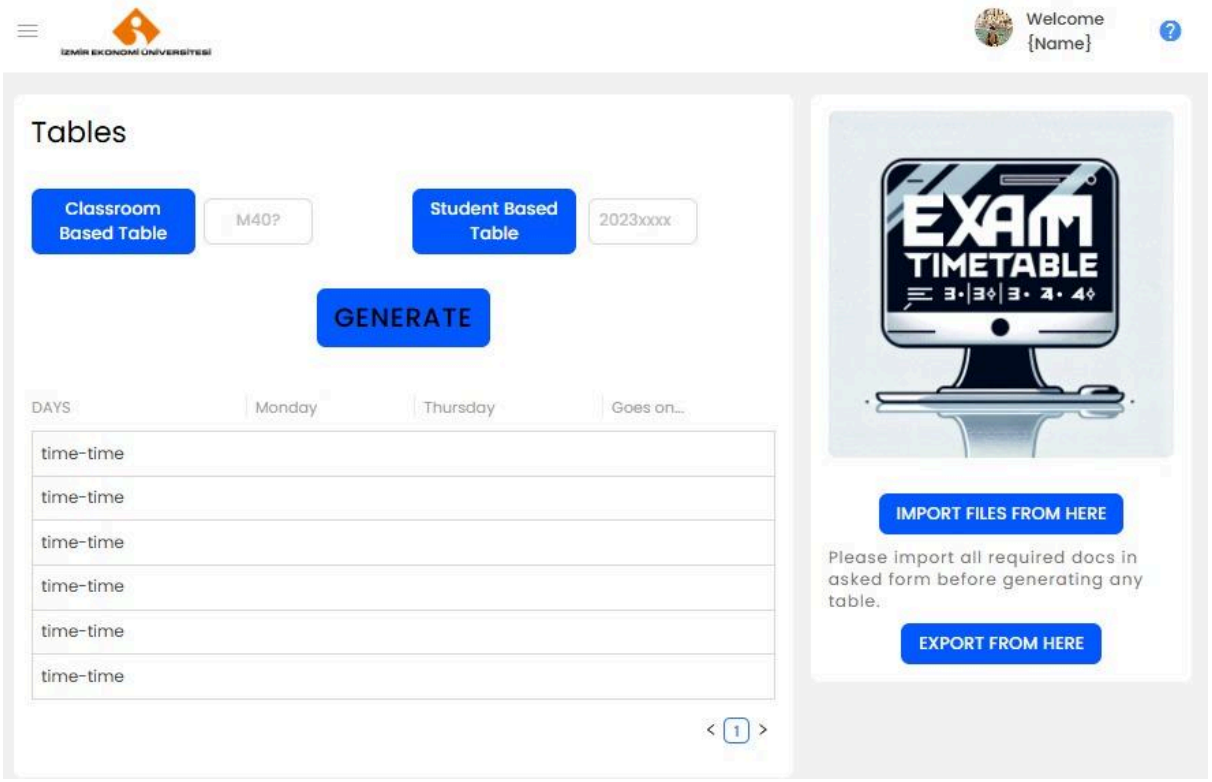
- **IMPORT FILES FROM HERE:** A prominent blue button that duplicates the Import Data menu functionality, providing an immediate call-to-action on the main screen.

- **EXPORT FROM HERE:** A corresponding blue button that duplicates the Export Schedule menu functionality. This button should remain disabled (greyed out) until a timetable has been successfully generated.

3.2.3 Timetable Display and Generation Controls

This section details the dynamic elements of the main screen where the user interacts with the generated schedule and controls the final output.

- **Table Selection Tabs:**
  - The interface must provide clear, distinct buttons to toggle the visualization of the final schedule. These tabs define the primary view for the generated timetable.
  - **Classroom Based Table:** Selecting this tab displays the schedule organized by **classroom**, showing which exam is held in which room at a specific time. A contextual identifier (e.g., M40?) may appear next to the button to indicate the currently viewed filter or room.
  - **Student Based Table:** Selecting this tab displays the schedule organized by **student**, allowing the user to view the exam conflicts and time slots assigned to individual students or cohorts.
- **Generation Button:**
  - **GENERATE:** This prominent button initiates the core scheduling algorithm. It must remain **disabled** until all required input files (Students, Classrooms, Courses) have been successfully imported and validated. Clicking this button transforms the "No data found" placeholder into the generated schedule.
- **Timetable Display Area:**
  - This central area is dedicated to visualizing the final timetable based on the selected **Table Selection Tab** (Classroom or Student Based).
  - **Initial State (Placeholder):** Before generation, this area displays a clear message such as **"No data found"** along with a relevant visual cue (e.g., a search icon over a table) to indicate that processing has not yet occurred.
  - **Dynamic Structure:** The visualization must clearly present the schedule over sequential **DAYS** (e.g., Monday, Thursday, Goes on...). The content (exam slots, courses) within the schedule grid changes dynamically based on the currently selected table view.



3.3 System Feedback and Error Handling

To maximize usability, the system must provide continuous and clear feedback:

- **Progress Indicators:** During resource-intensive operations (importing large files or generating the schedule), a non-intrusive progress bar or status indicator must be displayed to inform the user of the elapsed time and remaining duration.
- **Validation Alerts:** If a user attempts to generate a schedule without all required files loaded, a clear modal error message must appear, listing the missing files and directing the user back to the Import Data step.
- **Success Notifications:** Upon successful completion of an import or export, a brief, green-colored notification (e.g., a toast message) should confirm the action (e.g., "3 Files Imported Successfully" or "Timetable Saved to Desktop").

3.4 Help Menu and System Information

The **Help Menu** is a critical component of the user interface, dedicated to providing immediate access to support documentation and system information without requiring the user to leave the application environment. This feature significantly enhances the application's overall usability and self-service capability.

Help Menu Items

Help Documentation:

- **Purpose:** This function provides access to the system's full, contextual User Manual and support documentation, fulfilling Functional Requirement.
- **Display Mechanism:** The documentation must be displayed in a non-modal, side-panel interface (e.g., toast message) to ensure the user can reference instructions without losing sight of the main application window.
- **Content Requirement:** The documentation must cover system workflow, data requirements (including CSV format specifications), and common troubleshooting procedures.

About Application:

- **Purpose:** This item is mandatory for displaying technical metadata about the application instance.
- **Information Displayed:** The resulting dialog must clearly present:
  - The current **Application Version Number**.
  - The **Build/Release Date** of the installed software.
  - Official Copyright and Licensing statements.
- **Significance:** This information is crucial for developers and support personnel when debugging reported issues, ensuring the user is running the intended version.

Support Contact: (Added Detail)

- **Purpose:** Provides users with the necessary information to contact technical support or system administrators.
- **Content:** Display the relevant support email address and/or a link to an external support portal.