SD^2 - Class Room Assignment Manager

1. Project Overview

This section gives context for the reader so that they can understand the design that is being presented. It should describe the problem that your software attempts to address, identifying stakeholders and their stake in the system, and how your software will address the problem. This section gives context for the reader so that they can understand the design that is being presented. It should describe the problem that your software attempts to address, identifying stakeholders and their stake in the system, and how your software will address the problem. In this section provide the General Model including (Context Diagram and Use-case Diagram), user stories, product backlog as well as the main use-cases presenting the technical details based on requirement engineering.

The Class Room Assignment Manager (CRAM) is a web-based tool designed for schools, educational administrators, and scheduling managers. Its primary purpose is to facilitate class scheduling by allowing administrators to manage class details, enrollments, and automatically assign classrooms that best fit a course's size requirements. The stakeholders include:

- **Administrators:** They can add and manage user accounts (admin, professor, student), add classes either manually or through Excel file upload, upload or manage the available classrooms (including their capacities), and review overall scheduling.
- **Professors:** They access their dashboard to view, select, and update the courses they will be teaching, as well as review their teaching assignments.
- **Students:** They can view available classes, enroll in courses given there are no scheduling or capacity conflicts, and later view or unenroll from courses they no longer wish to attend.

General Model:

- **Context Diagram:** CRAM sits at the center with interfaces to users (admin, professor, student) and supports two major data domains: class scheduling and room management. External systems could include a student database, a staff portal, or external notification systems (if expanded).

- **Use-Case Diagram:** Main use cases include “Login”, “Manage Accounts”, “Add/Upload Classes”, “Manage Rooms”, “Assign Room Automatically”, “Enroll in Class”, “View/Unenroll Enrollment”, “Update Teaching Assignments”, etc.

- **User Stories:**

- As an admin, I want to upload an Excel file with room details so that the system can automatically assign classes to the most appropriate room.

- As an admin, I want to add classes manually, ensuring that each class is assigned a room based on the class size and available classroom capacity.
- As a professor, I want to view the classes I am teaching so that I can manage and update my teaching schedule.
- As a student, I want to view all available classes and enroll or unenroll based on my schedule constraints.

Product Backlog: Items include features for role-based login and dashboards, Excel file upload and preview for classes and rooms, auto-assignment algorithm for room selection, search and filtering capability for course listings, and account and class management tools.

Main Use-Cases:

- Class Creation and Room Assignment: Admin adds a class (or uploads a class Excel file). The system reads the class capacity and uses the auto-assignment algorithm to determine the best-fit room.
- Room Management: Admin creates or uploads classroom data (including room names/numbers and capacities), which is then used for matching courses during scheduling.
- User Account Management: Admin can add, edit, and remove user accounts; removing a user cascades to update enrollments (for students) or remove teaching assignments (for professors).

2. Architectural Overview

The introduction to this section should provide an overview of your architectural design. This section should begin by discussing any alternative designs that were examined. You should discuss why the alternatives were eliminated in favor of the architecture that you selected. In other words, you should discuss the rationale for choosing among architectural alternatives.
The current implementation of CRAM utilizes a monolithic architecture, serving an html page and running a local javascript app which manages the backend alongside a Mongo Database. Other architectures looked at include a more modular architecture utilizing the MERN stack.

2.1 Subsystem Architecture

This section should include a subsystem dependency diagram (i.e., a UML package diagram that shows dependency relationships between packages). Typically, your dependency diagram will have four layers: application-specific, application-generic, middleware, and system-software layers (much like the example presented in the slides in class).
For each package, describe its responsibility in the system. Also, identify and describe any architectural styles that are applied in your design, and explain why you chose to apply them (e.g., what properties do they offer that are desirable in your system, what issues do they address).

For reference material, see architectural styles on the web, revisit the lecture slides, and read the article on architectural styles posted on Canvas. For reference material on package diagrams, see the Sparx tutorial on UML package diagrams, the optional UML text listed on the syllabus, or visit www.uml.org.

The current implementation of CRAM employs a monolithic architecture with a single-page HTML/JavaScript application that uses browser-based localStorage for data persistence. Although we considered a modular approach using the full MERN stack (MongoDB, Express, React, and Node.js) for scalability and robustness, the monolithic approach was selected for its simplicity in a demo and educational setting.

Subsystems include:

- **Presentation Layer:** HTML/JavaScript interface that dynamically renders forms and dashboards based on user roles.
- **Application Logic:** JavaScript modules for user authentication, class management, room assignment, and Excel file processing (using SheetJS).
- **Data Persistence Layer:** A simplified storage model based on browser localStorage (in production, this would be replaced by a robust back-end database such as MongoDB or a relational database).
- **Excel Processing Module:** Integrates SheetJS to handle the upload and processing of Excel files for both classes and rooms.

The chosen architectural style is event-driven, as the system waits for user actions (such as clicks on buttons to upload files, manage classes, or enroll in courses) and processes these events accordingly. This design supports modularity within a monolithic application and has proven sufficient to address the core needs of this demo project.

2.2 Deployment Architecture

This section should include a UML Deployment Diagram (and supporting discussion) that includes a mapping of subsystems to deployment. The UML deployment diagram should show how the major components of your system will be assigned to different processors or different computers, and explain how those computers are connected (e.g., wireless, Ethernet LAN, Internet). Also, describe the communication protocol used (e.g., plain Java sockets, Java RMI, Java JDBC, HTTP, etc.) and why.

If you have a single-threaded software system that runs on a local device and does not connect to any other systems running on different computers across a network, then you do not need to complete this section – just write a single sentence stating “This software will run on a single processor.”

For references on UML deployment diagrams, see the Sparx tutorial or the optional UML text listed on the syllabus, or visit www.uml.org.
CRAM is currently designed to run on a single processor (i.e., it is fully client-side for demonstration purposes). For a production environment, the system could be deployed using a client-server model over HTTP with a back-end API server. However, in this demo, the HTML/JavaScript application runs locally with no external network dependency.

2.3 Persistent Data Storage

This section should identify what pieces of information must be stored and your approach to storing data (e.g., flat files, relational database). If you are using a flat file, you will identify the file format (what are the elements stored in the file, how are they separated, how do you represent the file). If you are using a database for persistent storage, give the database schema (i.e., describe the tables and their columns).

If your system does not need to store any data after a complete execution of the system (e.g., after the user exits), then you do not need to complete this section – just write a single sentence stating “This software does not require persistent data storage.”

The system uses browser localStorage for persistent data storage. The following information is stored:

- ***User Accounts:*** *Stored as a JSON array including username, password, and role.*
- ***Classes:*** *Stored as a JSON array where each class record includes class details, assigned room, capacity, enrolled student list, and professor assignment.*
- ***Rooms:*** *Stored as a JSON array with fields for room name (or number) and capacity.*
- ***Enrollments:*** *Stored as an object mapping student usernames to an array of enrolled class IDs.*

In a production system, this would be replaced by a database with appropriate schema definitions (for example, a MongoDB collection or a relational database with tables for users, classes, rooms, and enrollments).

2.4 Global Control Flow

This section should describe assumptions about how your system’s execution is controlled. You should identify any of the following models of control flow, and explain how they are applied (it is perfectly acceptable that you apply more than one of these):

- Procedural or event-driven?: Is your system procedure-driven and executes in a “linear” fashion, where every user every time has to go through the same steps, or is it an event-driven system that waits in a loop for events, and every user can generate the actions in a different order?
- Time dependency?: Do you have any timer-controlled actions in your system? Is your system of event response type, with no concern for real time, or is it a real-time system? If it is real-time, is it periodic, and what are the time constraints for each period?
- Concurrency?: Does your system use multiple threads? If so, identify the components that have separate threads of control and describe how the threads are synchronized.

CRAM is structured as an event-driven system. It listens to and responds to user events (such as form submissions, button clicks, and file uploads) rather than following a strict linear procedural order. There are no real-time constraints; the system is designed for interactive operation, and it does not use concurrency or multiple threads (all operations occur in the client’s browser).

3. Detailed System Design

This section presents a detailed design for the architecture described in the Architectural Overview, and should be consistent with the architectural styles and package diagram identified in Sections 1 and 2 of your documents. For each major part (i.e., component, module, or package) identified in your architecture, create a subsection. In each subsection that corresponds to a component or package, you will model the static view with UML class diagrams and the dynamic view with UML sequence diagrams:

3.1 Static View

You must include UML class diagrams showing further decomposition of the major modules and the relationships among these classes. In each class in the UML class diagram, you must show:

- Important attributes, their type, and their visibility
- Important operation/method names, their parameters, return types, and their visibility (public, private, protected, package)
- Associations between classes and multiplicity constraints

In addition, you will also need to write support text that justifies your decomposition of your modules into the classes shown in your UML class diagram. This justification should discuss the responsibilities of each class. The justification should describe other alternative designs, if any, and why your design is better.
Finally, you should describe any design patterns that are included in this design and why you’ve applied them.
For reference material, see Robert Martin’s article on UML class diagrams, the Sparx tutorial on UML package and class diagrams, or the optional UML textbook recommended on the syllabus.

For reference material on design patterns, check out the POSA and GoF or this very handy online catalog of design patterns or this website on common patterns.

UserManager: Handles login, account creation, and removal.
Attributes: currentUser, accounts
Methods: login(), addAccount(), removeAccount(), loadDashboard()

ClassManager: Manages class operations (e.g., create, edit, delete, and Excel upload processing).
Attributes: classes, enrollments
Methods: addClass(), updateClass(), deleteClass(), enrollInClass(), unenrollFromClass(), uploadExcelClasses()

RoomManager: Manages available classrooms including manual entry and Excel upload processing as well as room assignment logic.
Attributes: rooms
Methods: addRoom(), updateRoom(), deleteRoom(), uploadExcelRooms(), assignRoom(requiredCapacity)

DashboardRenderer: Responsible for rendering views based on role (Admin, Professor, Student).
Methods: renderAdminDashboard(), renderProfessorDashboard(), renderStudentDashboard(), renderSearch(), etc.

Justification:
This decomposition separates concerns clearly: user-related operations are isolated from class scheduling and room assignment logic. The design avoids tight coupling by using a shared, simple data storage mechanism (localStorage) and clearly defining module responsibilities. Design patterns such as “Facade” (DashboardRenderer acting as a unified interface to the underlying subsystems) and “Singleton” (global access to localStorage via utility functions) have been applied.

3.2 Dynamic View

You must show the design of your system’s behavior using UML sequence diagrams. These sequence diagrams should show the time-ordered sequence of interactions among classes to support an important system function. Your sequence diagrams should be consistent with the class diagrams given in Section 3.1. In other words, you should not have participating objects in an interaction that do not appear in a class diagram; if you find that this is the case, you should go back and revise your class diagram to include the new element. You may supplement your sequence diagrams with state-transition diagrams or activity diagrams (useful for describing algorithms), but these are not required.

Make sure to add your Testplans, previous and current sprints and brief sprint reviews as well as any additional items you feel are helpful at any point. Remember, it's not necessary to add documents that are not providing useful information. Only necessary items. You can add/subtract items or update them using version control to your design.

User Login Sequence:

1. User enters credentials and clicks "Login."
2. UserManager verifies credentials (via localStorage), then DashboardRenderer loads the appropriate dashboard (admin, professor, or student).

Class Creation and Room Assignment Sequence:

1. Admin fills out the "Add Class" form with class details and required capacity.
2. ClassManager calls RoomManager.assignRoom(requiredCapacity) to select the best-fit room.
3. The class data is stored in localStorage, and a confirmation message with the assigned room is displayed.

Excel Upload Sequence for Rooms and Classes:

1. Admin selects an Excel file and clicks "Upload."
2. The Excel Processing module (using SheetJS) parses the file and converts it to JSON.
3. A preview of data is rendered, and upon confirmation, RoomManager or ClassManager iterates over the JSON rows to create new records with automatic room assignment (for classes) or simple insertion (for rooms).

Enrollment Sequence (for Students):

1. Student selects a class and clicks "Enroll."
2. ClassManager checks capacity and any scheduling conflicts.
3. If no conflict is detected, the student is added to the class's enrolled list, and the enrollment is stored in localStorage.

Test Plans:

- **Login Tests:** Verify that valid accounts log in and invalid credentials are rejected.
- **Classroom Assignment Tests:** Test various class capacities against a range of room capacities to ensure the best-fit room is assigned.
- **Excel Upload Tests:** Validate that file parsing correctly interprets column headers and data, then updates the system storage appropriately.
- **Enrollment Conflict Tests:** Check that students cannot enroll in overlapping classes.

Sprint Reviews & Product Backlog:

- Sprint 1: Create basic login and dashboard functionality.
- Sprint 2: Implement class and room management modules with manual input forms.
- Sprint 3: Integrate Excel upload features using SheetJS and implement automatic room matching.
- Sprint 4: Enhance search, filtering, and user management capabilities; perform integration and system testing.