C-AI311 – Deliverable #3 Design Report

Automated University Garage Management System (AUGMS)

Course: C-SW312: Introduction to Software Engineering

Semester: Fall 2025

Submission Date: December 27, 2025

Group Members: Aly Hassan, Mohamed Ehab, Alaa Shaaban, Kenzy Zedan, Shady Ashraf, Youssef Osama.

Abstract

This technical report documents our work for Deliverable #5 of the C-SW312 project. The deliverable focuses on object-oriented design driven by use cases. We build upon previous deliverables, including the use case diagram, system sequence diagrams (SSDs), activity diagrams, and domain model class diagram.

The report is divided into two main parts: Modelling and Implementation. In the Modelling Part, we selected core use cases, then classified them based on complexity, perform step-by-step diagrams for each, and developed a design class diagram and a package diagram. In the Implementation Part, we transform the design into Java code using StarUML java tool, producing a walking skeleton with a three-layered architecture.

1. Introduction

1.1 Project Overview

The Automated University Garage Management System (AUGMS) is a software solution designed to streamline parking operations at a university campus. It handles vehicle registration, entry and exit validation, occupancy monitoring, user access management, service requests (e.g., EV charging, cleaning), sensor error alerts, reporting, and more. The system integrates hardware components like sensors, gates, and displays with software layers for user interaction, business logic, and data persistence. Key stakeholders include students/faculty (parking users), garage admins, university managers, and system sensors.

The objectives of AUGMS are to:

- Automate vehicle entry/exit to reduce manual intervention.

- Provide real-time occupancy monitoring and alerts.
- Manage user registrations, access, and service requests efficiently.
- Generate reports for usage analysis and error logging.
- Ensure security, reliability, and scalability.

1.2 Assumptions

- The system uses a three-layered architecture: View (presentation), Controller (logic mediation), and Domain/Data (business logic and persistence).
- All use cases assume a relational database for persistence, with DAOs handling CRUD operations.
- User authentication is required for admin and manager actions; we assume a basic credential validation mechanism.
- Real-time updates (e.g., occupancy) use push notifications or polling; we assume reliable network connectivity.
- Fees for services are calculated based on type; exact formulas are simplified
- The domain model from previous deliverables includes core classes like ParkingUser, Vehicle, ParkingGarage, ParkingSpot, etc.

1.3 Rationale for Design Approach

Our design is use-case driven, ensuring traceability from analysis to design. We prioritize separation of concerns, low coupling, and high cohesion. The first-cut DCD starts from the domain model, adding controllers and view classes. Techniques like CRC help identify responsibilities collaboratively, while diagrams visualize interactions.

2. Classification of Use Cases

Based on the use case diagram from prior deliverables, we classified use cases by complexity:

- **Simple**: Involve basic operations with minimal branching or layers (e.g., data retrieval/update with little validation).
    - UC-104: View Parking Status
    - UC-106: Register User Account
    - UC-107: Update Vehicle Details
    - UC-503: Generate Sensor Error Alert

- **Moderate**: Require coordination across layers with some validation and alternatives.
    - UC-101: Register Vehicle
    - UC-103: Process Vehicle Exit
    - UC-302: Manage User Access

    - UC-201: Manage Vehicle Registration
- **Complex**: Involve multiple actors, conditional flows, real-time elements, or integrations (e.g., payments, sensors).
    - UC-102: Validate Vehicle Entry
    - UC-105: Request Service
    - UC-202: Monitor Garage Occupancy
    - UC-501: Generate Usage Report (assumed complex due to data aggregation)

Rationale: Classification considers interaction count, alternatives, and layer involvement. We selected core use cases representing system functionality.

3. Use Case Realization

3.1 Simple Use Cases

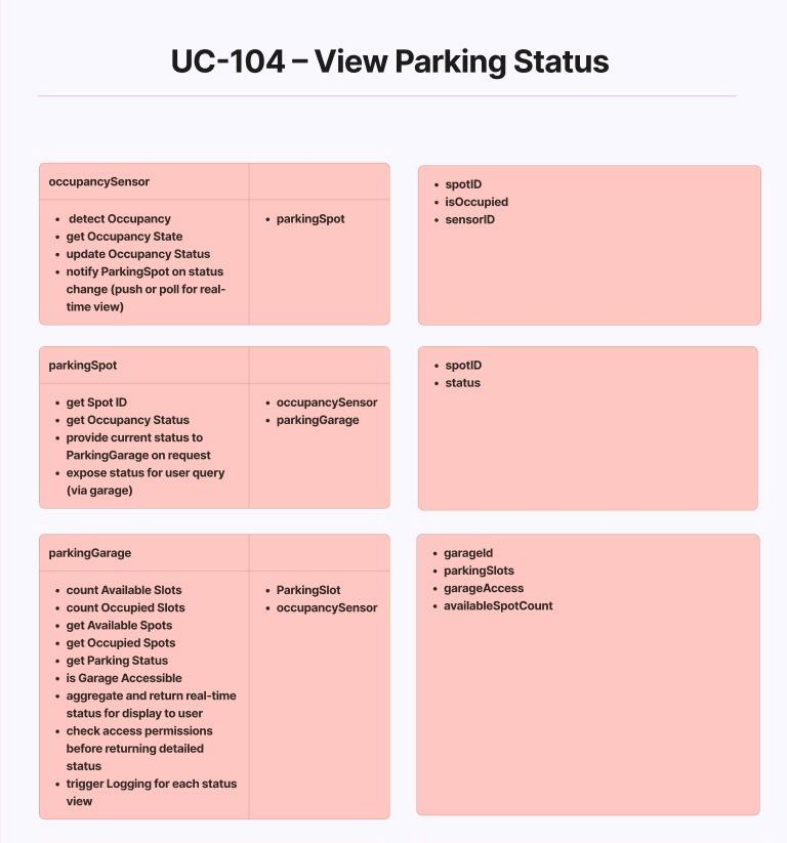
3.1.1 UC-104: View Parking Status

Description: Users view current parking spot statuses, counts.

Step-by-Step First-Cut DCD Partition:

1. Start with domain classes: ParkingGarage, ParkingSpot, OccupancySensor.
2. Add view class: ParkingStatusView (handles display).
3. Add controller: ParkingStatusController (mediates requests).
4. Add data access: ParkingSpotDAO, OccupancySensorDAO.

5. Identify associations: ParkingGarage aggregates ParkingSpot; ParkingSpot links to OccupancySensor.



CRC Technique:

- occupancySensor: Responsibilities - detect Occupancy, get Occupancy State, update Occupancy Status, notify ParkingSpot on status change (push or pull for real-time view). Collaborators - parkingSpot.
- parkingSpot: Responsibilities - get Spot ID, provide Current Status to ParkingGarage on request, expose status for user query (via garage). Collaborators - occupancySensor, parkingGarage.
- parkingGarage: Responsibilities - count Available Slots, count Occupied Slots, get Available Spots, get Occupied Spots, is Garage Accessible, aggregate and return real-time status for display to user, check access permissions before returning detailed status. Collaborators - ParkingSpot, occupancySensor.

Rationale: CRC identifies clear responsibilities, assuming real-time polling for status.

Added methods: getSpotID(), isOccupied(), sensorID() to classes.

3.1.2 UC-106: Register User Account

Description: Users register accounts with personal details.

Step-by-Step First-Cut DCD Partition:

1. Domain: User (Student/Faculty), GarageSystem.
2. View: UserRegistrationView.
3. Controller: RegistrationController.
4. Data: ParkingUserDAO.
5. Associations: User links to GarageSystem.

CRC Technique:

- User (Student/Faculty): Responsibilities - provide personal information, submit registration request, receive registration result (success/pending/error). Collaborators - GarageSystem.

- GarageSystem: Responsibilities - receive user registration request, validate input data, create user account, set registration status, notify user of result. Collaborators - User, Logging.
- Logging: Responsibilities - record user registration event, store registration timestamp, retrieve logs by type. Collaborators - GarageSystem, User.

Rationale: Simple validation assumed (e.g., unique email); no complex approvals.

Added: userID, name, email, role. Methods: createUserAccount(), validateInputData().

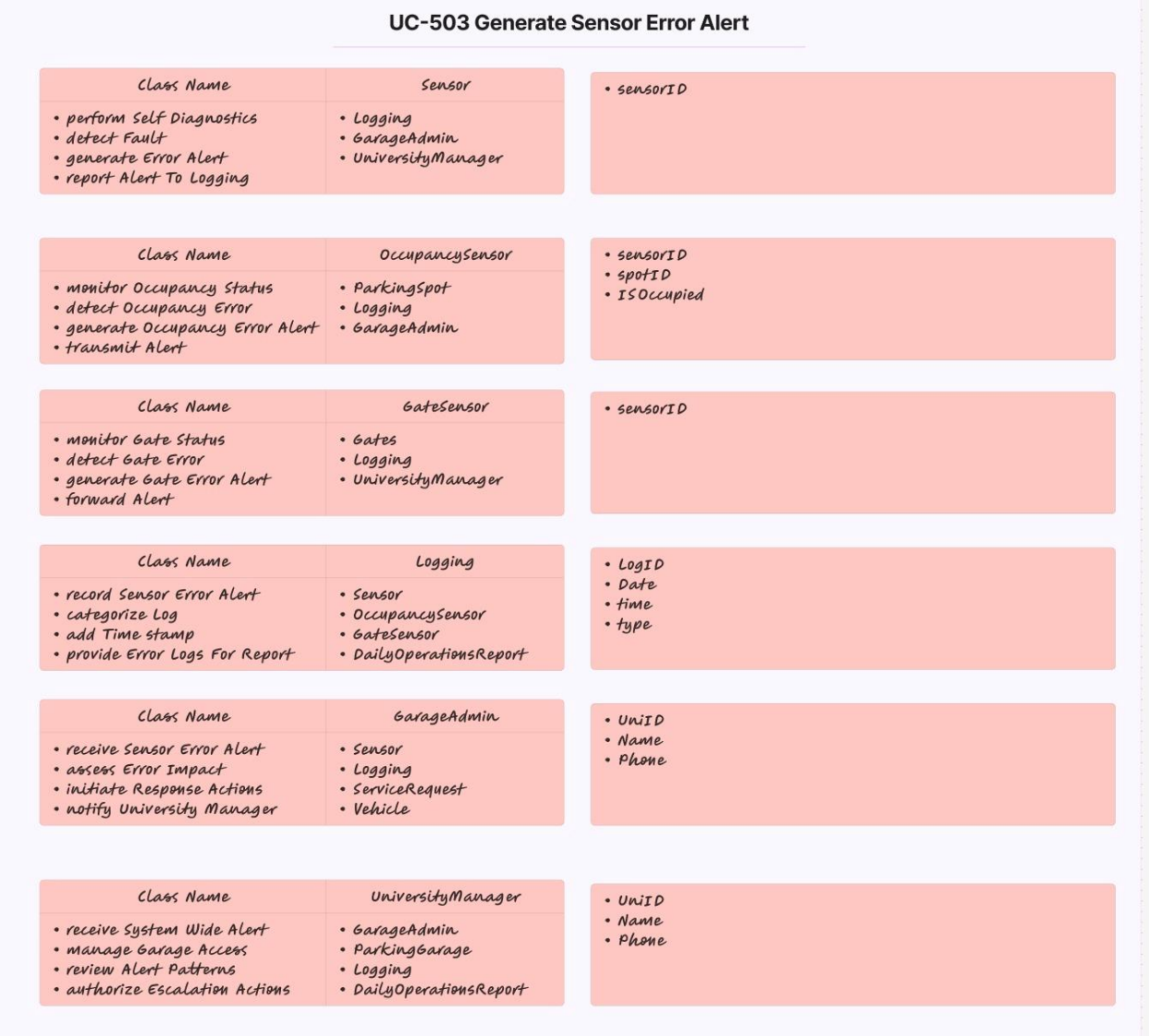
3.1.3 UC-503: Generate Sensor Error Alert

Description: System generates alerts for sensor faults.

Step-by-Step First-Cut DCD Partition:

1. Domain: Sensor, OccupancySensor, GateSensor, GarageAdmin, UniversityManager, Logging.
2. View: SensorAlertView.
3. Controller: SensorAlertController.
4. Data: SensorDAO, OccupancySensorDAO, GateSensorDAO.

5. Associations: Sensors link to Logging and Admins.



CRC Technique:

- Sensor: Responsibilities - perform Self Diagnostics, detect Fault, generate Error Alert, report Alert To Logging. Collaborators - GarageAdmin, UniversityManager.
- OccupancySensor: Responsibilities - monitor Occupancy Status, detect Occupancy Error, transmit Occupancy Error Alert. Collaborators - ParkingSpot, Logging, GarageAdmin.
- GateSensor: Responsibilities - monitor Gate Status, detect Gate Error, forward Gate Error Alert. Collaborators - Logging, UniversityManager.

- Logging: Responsibilities - record Sensor Error Alert, categorize Log, add Time Stamp, provide Error Logs For Report. Collaborators - Sensor, OccupancySensor, GateSensor.
- GarageAdmin: Responsibilities - receive Sensor Error Alert, assess Error Impact, initiate Response Actions, notify University Manager. Collaborators - Sensor, Logging.
- UniversityManager: Responsibilities - receive System Wide Alert, manage Garage Access, review Alert Patterns, authorize Escalation Actions. Collaborators - GarageAdmin, Logging.

Rationale: Assumes automatic fault detection; alerts via view layer.

Added: sensorID, spotID, isOccupied. Methods: detectFault(), generateAlert().

3.1.4 UC-107: Update Vehicle Details

Description: Users update vehicle info like plate, model.

Step-by-Step First-Cut DCD Partition:

1. Domain: Vehicle, Logging.
2. View: VehicleUpdateView.
3. Controller: VehicleUpdateController.
4. Data: VehicleDAO.
5. Associations: Vehicle links to ParkingUser.

CRC Technique:

- Vehicle: Responsibilities - store vehicle information (license plate, model, color, access status), update vehicle details, validate updated vehicle data, save updated vehicle information, ensure the vehicle belongs to the authenticated user. Collaborators - Parking User, Authentication Service, Vehicle Service, Database.
- Logging: Responsibilities - record every update action, store timestamp, user, and vehicle involved, keep activity history for reporting, provide data for admin reports. Collaborators - Vehicle Service, Authentication Service, Database, Admin/Reporting Service.

Rationale: Authentication assumed to prevent unauthorized updates.

Added: vehicleID, licensePlate, model, color, accessStatus, userID. Methods: updateDetails(), validateData().

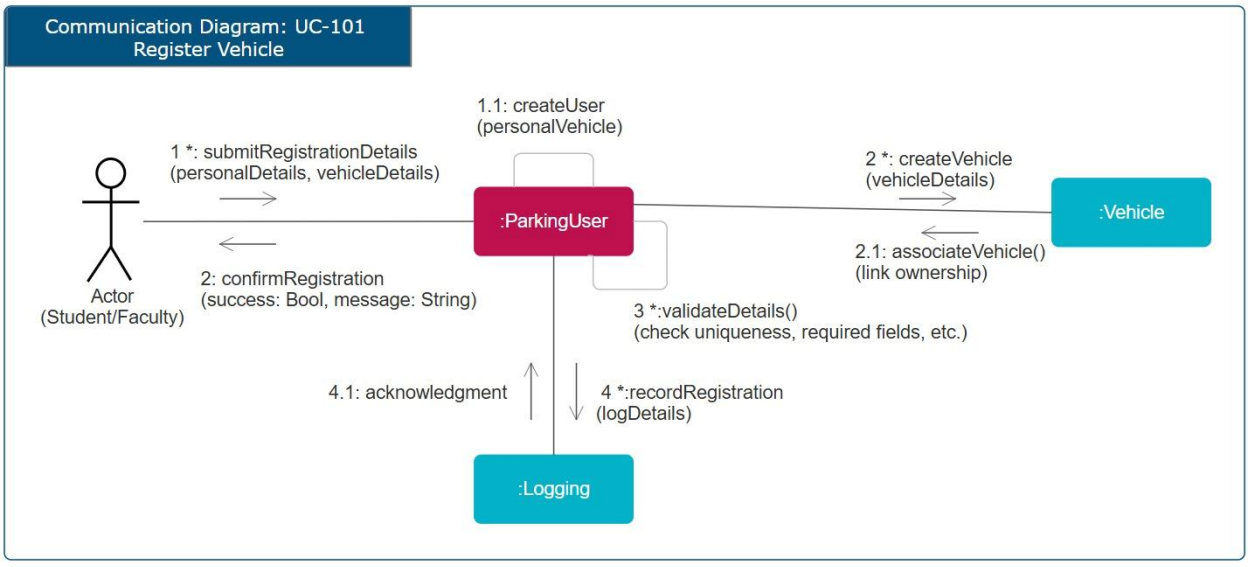
3.2 Moderate Use Cases

3.2.1 UC-101: Register Vehicle

Description: Users register vehicles with personal/vehicle details.

Step-by-Step First-Cut DCD Partition:

1. Domain: ParkingUser, Vehicle, Logging.
2. View: RegistrationView.
3. Controller: RegistrationController.
4. Data: ParkingUserDAO, VehicleDAO, LoggingDAO.
5. Associations: ParkingUser owns Vehicle.



Multi-Layer Communication Diagram: The diagram shows: Actor (Student/Faculty) submits details to ParkingUser, which creates user and vehicle, validates uniqueness, records in Logging.

Rationale: Moderate due to validation and linking; assumes unique license plates.

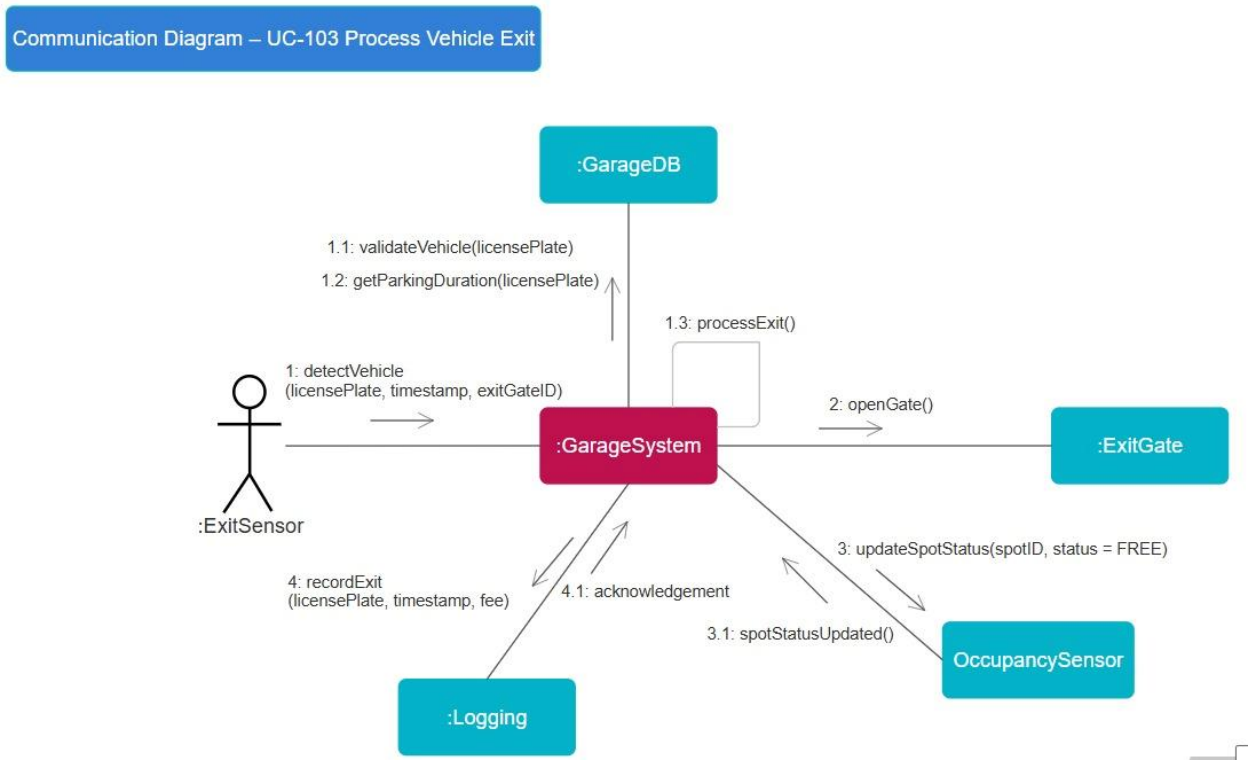
Added methods: submitRegistrationDetails(), createUser(), createVehicle(), associateVehicle(), validateDetails(), recordRegistration().

3.2.2 UC-103: Process Vehicle Exit

Description: Validate exiting vehicle, calculate fee, update spot.

Step-by-Step First-Cut DCD Partition:

1. Domain: GarageSystem, ExitSensor, GarageDB, ExitGate, Logging, OccupancySensor.
2. View: VehicleExitView.
3. Controller: VehicleExitController.
4. Data: VehicleDAO, ParkingSpotDAO.
5. Associations: ExitSensor detects Vehicle; GarageSystem processes.



Multi-Layer Communication Diagram: Diagram: ExitSensor detects vehicle, sends to GarageSystem, which validates, gets duration from GarageDB, processes exit, updates spot, opens gate, records in Logging.

Rationale: Assumes timestamp-based fee calculation; garage not full on exit.

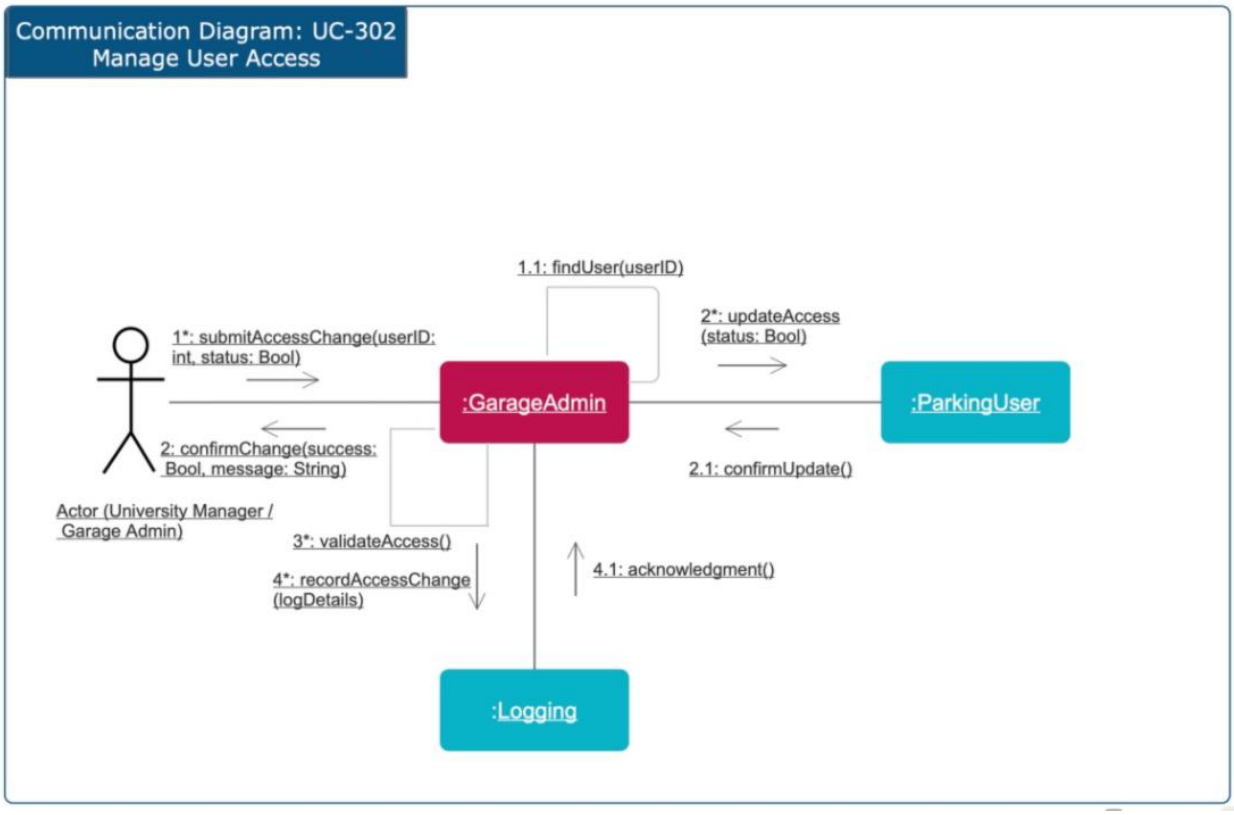
Added: licensePlate, timestamp, exitGateID. Methods: detectVehicle(), validateVehicle(), getParkingDuration(), processExit(), updateSpotStatus(), recordExit().

3.2.3 UC-302: Manage User Access

Description: Admins change user access status.

Step-by-Step First-Cut DCD Partition:

1. Domain: GarageAdmin, ParkingUser, Logging.
2. View: UserAccessView.
3. Controller: UserAccessController.
4. Data: GarageAdminDAO, ParkingUserDAO.
5. Associations: GarageAdmin updates ParkingUser.



Multi-Layer Communication Diagram: Diagram: Actor (University Manager/Garage Admin) submits change, GarageAdmin finds user, updates access, confirms, validates, records in Logging.

Rationale: Boolean status for access; assumes admin authentication.

Added: userID, status (Bool). Methods: submitAccessChange(), findUser(), updateAccess(), confirmUpdate(), validateAccess(), recordAccessChange().

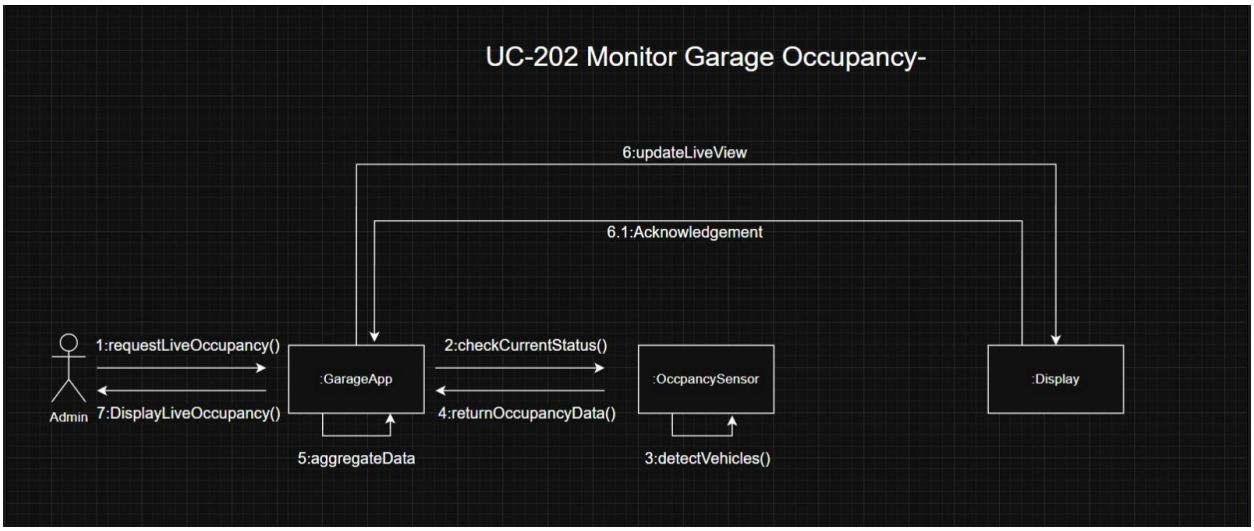
3.2.4 UC-202: Monitor Garage Occupancy

Description: Admins request live occupancy views.

Step-by-Step First-Cut DCD Partition:

1. Domain: GarageApp, OccupancySensor, Display.
2. View: ParkingStatusView.
3. Controller: ParkingStatusController.
4. Data: OccupancySensorDAO.

5. Associations: GarageApp requests from Sensor.



Multi-Layer Sequence Diagram: Admin requests live, GarageApp checks status, Sensor detects vehicles, aggregates data, returns data, displays, updates view, acknowledgment.

Rationale: Real-time assumption via aggregation; display is external hardware.

Added: Methods: requestLiveOccupancy(), checkCurrentStatus(), detectVehicles(), aggregateData(), returnOccupancyData(), displayLiveOccupancy(), updateLiveView().

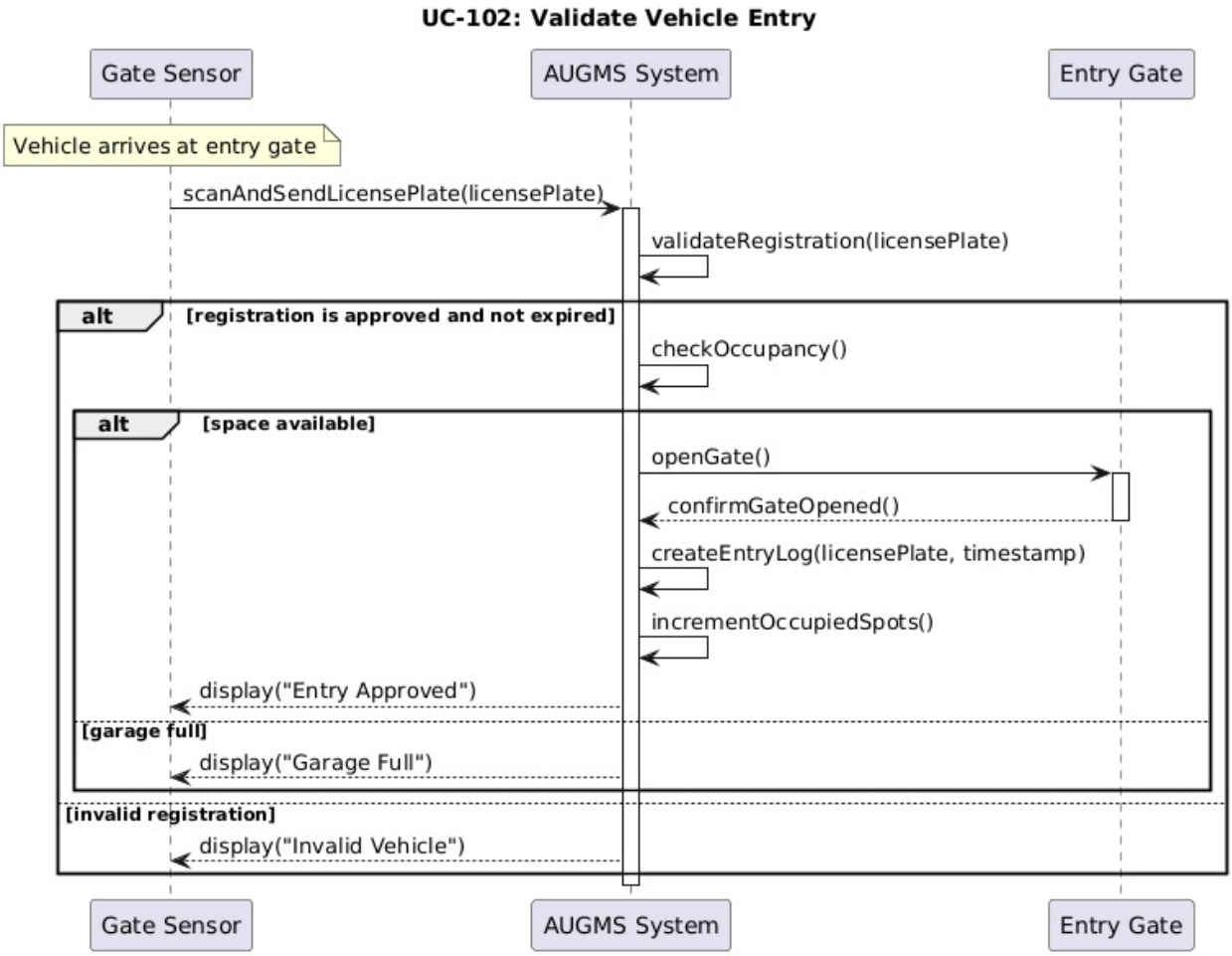
3.3 Complex Use Cases

3.3.1 UC-102: Validate Vehicle Entry

Description: Validate entering vehicle, check space, open gate.

Step-by-Step First-Cut DCD Partition:

1. Domain: AUGMSSystem, GateSensor, EntryGate.
2. View: VehicleEntryView.
3. Controller: VehicleEntryController.
4. Data: VehicleDAO, ParkingGarageDAO.
5. Associations: GateSensor scans to AUGMSSystem.



Multi-Layer Sequence Diagram: Vehicle arrives, GateSensor scans/sends plate, AUGMSSystem validates registration, checks occupancy, opens gate if space, creates log, increments occupied, displays status.

Rationale: Alternatives for invalid/full; assumes space check before entry.

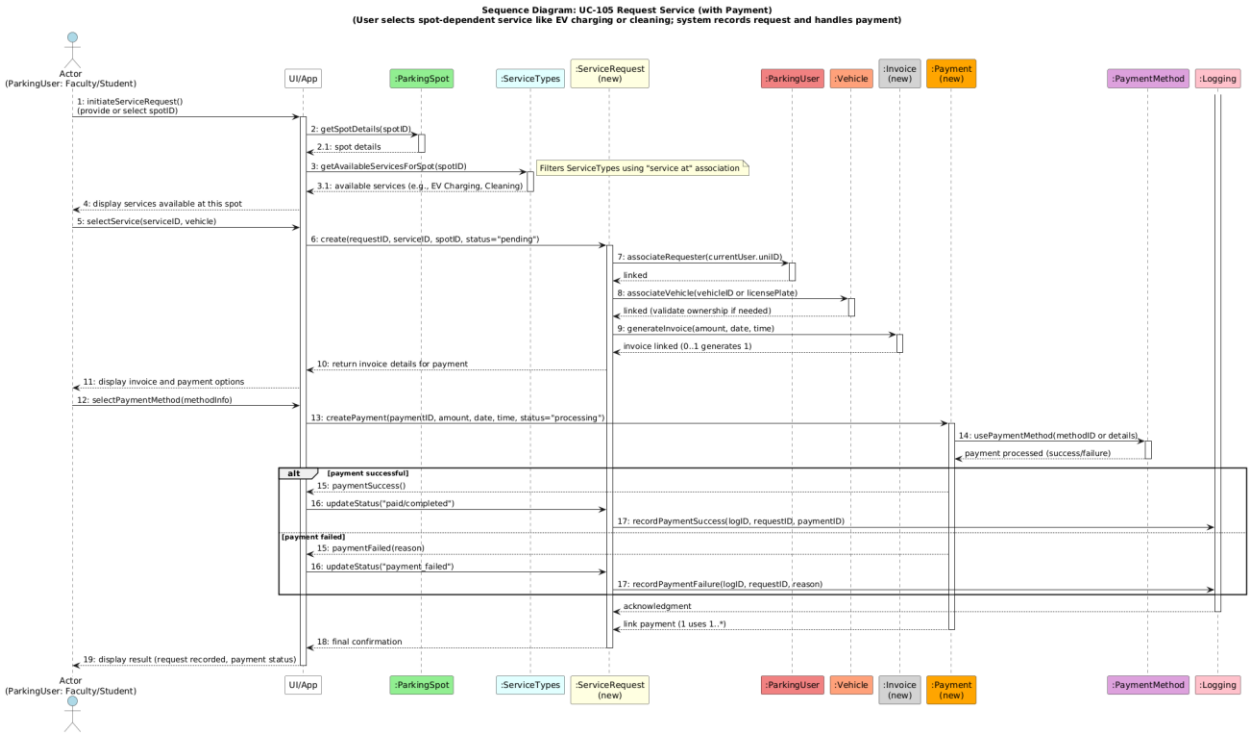
Added: licensePlate. Methods: scanAndSendLicensePlate(), validateRegistration(), checkOccupancy(), openGate(), confirmGateOpened(), createEntryLog(), incrementOccupiedSpots().

3.3.2 UC-105: Request Service

Description: Users request spot-dependent services with payment.

Step-by-Step First-Cut DCD Partition:

1. Domain: ParkingSpot, ServiceRequest, ParkingUser, Vehicle, Invoice, Payment, PaymentMethod, Logging.
2. View: ServiceRequestView.
3. Controller: ServiceRequestController.
4. Data: ServiceRequestDAO, InvoiceDAO, PaymentDAO.
5. Associations: ServiceRequest links to ParkingSpot, User, Vehicle, Invoice.



Multi-Layer Sequence Diagram: Actor initiates, gets details, available services, creates request, associates user/vehicle, generates invoice, displays payment, processes, records success/failure in Logging.

Rationale: Assumes service types (EV charging, cleaning); payment success updates status.

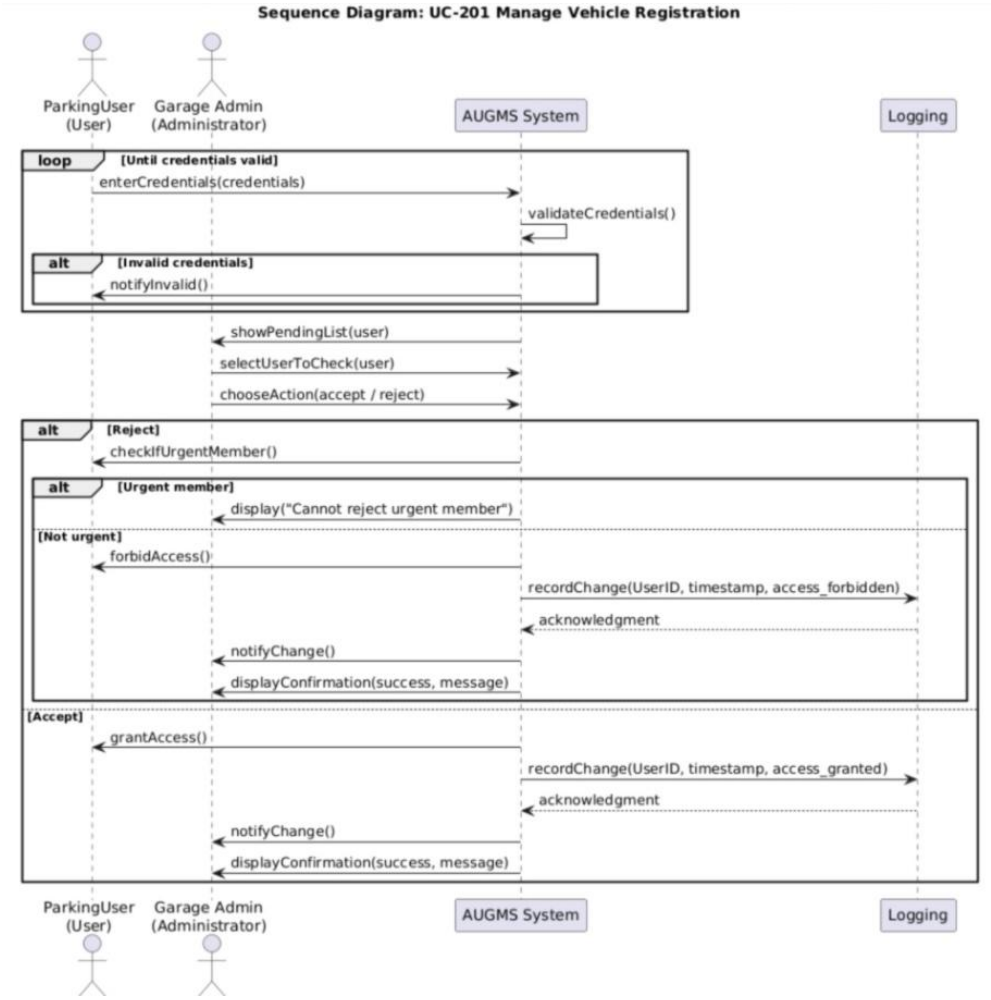
Added: spotID, serviceID, requestID, paymentID, amount, date, time. Methods: initiateServiceRequest(), getSpotDetails(), getAvailableServicesForSpot(), createRequest(), associateRequester(), associateVehicle(), generateInvoice(), createPayment(), usePaymentMethod(), recordPaymentSuccess/Failure().

3.3.3 UC-201: Manage Vehicle Registration

Description: Admins manage pending registrations.

Step-by-Step First-Cut DCD Partition:

1. Domain: ParkingUser, GarageAdmin, AUGMSSystem, Logging.
2. View: VehicleManagementView.
3. Controller: VehicleManagementController.
4. Data: ParkingUserDAO, VehicleDAO.
5. Associations: GarageAdmin reviews ParkingUser.



Multi-Layer Communication Diagram: (Note: Provided as sequence, but adapted.) Loop for credentials, show pending, select action (accept/reject), check urgent, forbid/record, notify.

Rationale: Loop for multiple reviews; urgent members can't be rejected.

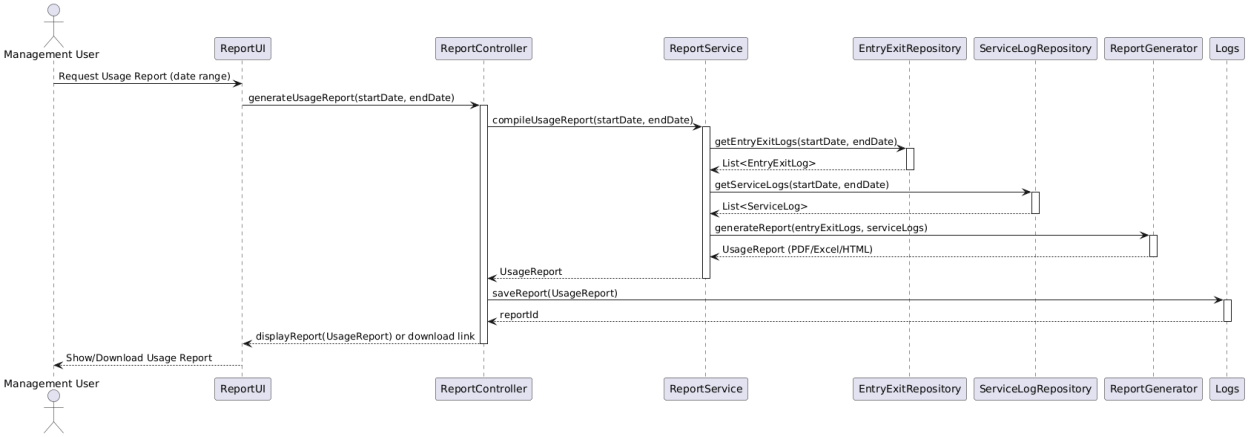
Added: credentials, userID, timestamp, access. Methods: validateCredentials(), showPendingList(), selectUserToCheck(), chooseAction(), checkIfUrgentMember(), recordChange(), notifyChange().

3.3.4 UC-301: Generate Usage Report

Description: Managers request usage reports.

Step-by-Step First-Cut DCD Partition:

1. Domain: ReportUI, ReportController, ReportService, EntryExitRepository, ServiceLogRepository, ReportGenerator, Logs.
2. View: ReportView.
3. Controller: ReportController.
4. Data: LogsDAO.
5. Associations: ReportService aggregates logs.



Multi-Layer Sequence Diagram: Management User requests report (date range), Controller compiles, Service gets logs, generates report, saves, displays/downloads.

Rationale: Assumes PDF/Excel/HTML formats; date range filtering.

Added: startDate, endDate. Methods: requestUsageReport(), generateUsageReport(), compileUsageReport(), getEntryExitLogs(), getServiceLogs(), generateReport(), saveReport(), displayReport().

4. Complete Design Class Diagram

The final design class diagram integrates all updated partitions from the 12 use cases. It includes view, controller, service/domain, entity, and data layers with methods, attributes, and relationships (associations, inheritance, dependencies).

Rationale: Layered for MVC pattern; inheritance for sensors; DAOs for persistence. Assumptions: All entities have IDs; services handle business logic.

5. Package Diagram

The package diagram organizes the complete DCD into layers and subsystems for modularity.

- View Layer: RegistrationView, ParkingStatusView, etc.
- Controller Layer: RegistrationController, ParkingStatusController, etc.
- Domain Layer: ParkingUser, Vehicle, ParkingGarage, etc.
- Data Layer: ParkingUserDAO, VehicleDAO, etc.
- Entity: Sensor, Logging, etc.
- Service: UserRegistrationService, VehicleEntryService, etc.

Rationale: Packages reduce dependencies; e.g., View depends on Controller, which depends on Service/Domain. Assumptions: No cyclic dependencies; services can depend on each other for composition.

6. Final Architecture:

parking-system/

├── main.java

├── ApplicationConfig.java

│

├── presentation/

│ ├── RegistrationView.java

│ ├── ParkingStatusView.java

│ ├── ServiceRequestView.java

│ ├── VehicleManagementView.java

│ ├── UserAccessView.java

│ ├── VehicleUpdateView.java

```
│   ├── SensorAlertView.java
│   ├── ReportView.java
│   └── ManageServiceRequestView.java
│
├── controller/
│   ├── RegistrationController.java
│   ├── ParkingStatusController.java
│   ├── ServiceRequestController.java
│   ├── VehicleManagementController.java
│   ├── UserAccessController.java
│   ├── VehicleUpdateController.java
│   ├── SensorAlertController.java
│   ├── ReportController.java
│   └── ManageServiceRequestController.java
│
├── service/
│   ├── RegistrationService.java
│   ├── VehicleEntryService.java
│   ├── VehicleExitService.java
│   ├── ParkingStatusService.java
│   ├── ServiceRequestService.java
│   ├── VehicleManagementService.java
│   ├── UserAccessService.java
│   ├── VehicleUpdateService.java
│   ├── SensorAlertService.java
│   ├── ReportService.java
```

```
│   ├── InvoiceService.java
│   ├── PaymentService.java
│   ├── AuthenticationService.java
│   ├── LoggingService.java
│   ├── ReportGenerator.java
│   └── ManageServiceRequestService.java
│
├── dao/
│   ├── ParkingUserDAO.java
│   ├── VehicleDAO.java
│   ├── ParkingGarageDAO.java
│   ├── ParkingSpotDAO.java
│   ├── OccupancySensorDAO.java
│   ├── EntryGateDAO.java
│   ├── ExitGateDAO.java
│   ├── GateSensorDAO.java
│   ├── SensorDAO.java
│   ├── ServiceRequestDAO.java
│   ├── ServiceTypeDAO.java
│   ├── InvoiceDAO.java
│   ├── PaymentDAO.java
│   ├── PaymentMethodDAO.java
│   ├── GarageAdminDAO.java
│   ├── UniversityManagerDAO.java
│   ├── LoggingDAO.java
│   ├── EntryExitRepository.java
```

```
│   ├── ServiceLogRepository.java
│   └── ParkingStatusDAO.java
│
├── entity/
│   ├── ParkingUser.java
│   ├── Vehicle.java
│   ├── ParkingGarage.java
│   ├── ParkingSpot.java
│   ├── OccupancySensor.java
│   ├── Sensor.java
│   ├── GateSensor.java
│   ├── EntryGate.java
│   ├── ExitGate.java
│   ├── ServiceRequest.java
│   ├── ServiceType.java
│   ├── Invoice.java
│   ├── Payment.java
│   ├── PaymentMethod.java
│   ├── GarageAdmin.java
│   ├── UniversityManager.java
│   ├── Log.java
│   ├── EntryExitLog.java
│   ├── ServiceLog.java
│   ├── ErrorLog.java
│   ├── UsageReport.java
│   └── ParkingStatus.java
```

```
│
└── util/
    ├── enums/
    ├── exceptions/
    └── helpers/
```

7. Prototype Implementation (Walking Skeleton and Selected Use Cases)

We extracted a walking skeleton from the StarUML model and implementing a functional prototype in Java.

7.1 Technologies Used

- **Language**: Java 17
- **GUI Framework**: JavaFX (for responsive desktop views)
- **Database**: Microsoft SQL Server 2022 (hosted via SQL Server Management Studio)
- **Persistence**: JDBC with prepared statements (DAO pattern)
- **Build Tool**: Maven (for dependency management)
- **IDE/Modeling**: StarUML (code generation) + Visual Studio Code

7.2 Implemented Use Cases

We fully implemented and tested the following four use cases, covering core user and admin flows:

1. **UC-104: View Parking Status**
    - Real-time display of available/occupied spots, total counts, and garage accessibility.
    - JavaFX TableView populated from ParkingSpotDAO.
    - Refresh button triggers polling from database.
2. **UC-101: Register Vehicle**
    - User registration combined with vehicle registration in a multi-step JavaFX form.
    - Validation for unique license plates and required fields.
    - Success/failure messages displayed.
3. **UC-103: Process Vehicle Exit**
    - Simulated exit by entering license plate.
    - Validates registered vehicle, calculates parking duration and fee (simple rate-based), updates spot status to FREE, logs entry/exit.

- Displays calculated fee.

7.3 Key Implementation Notes

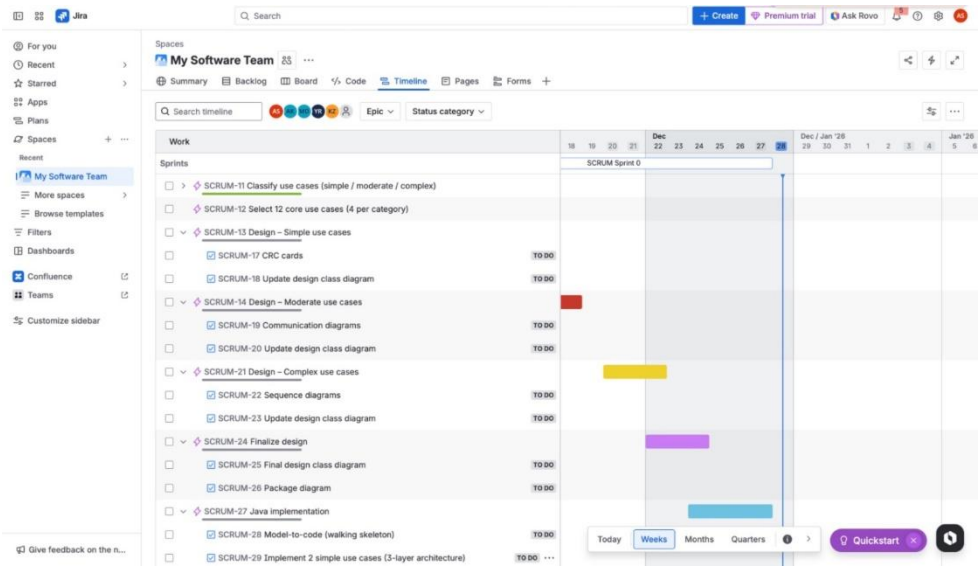
- Database schema matches entity classes (tables: Users, Vehicles, ParkingSpots, EntryExitLogs, etc.).
- All DAO methods use try-with-resources for safe resource management.
- Basic error handling and logging to console (to be replaced with full LoggingService later).
- The prototype is fully runnable and demonstrates end-to-end flow from GUI → Controller → Service → DAO → Database.

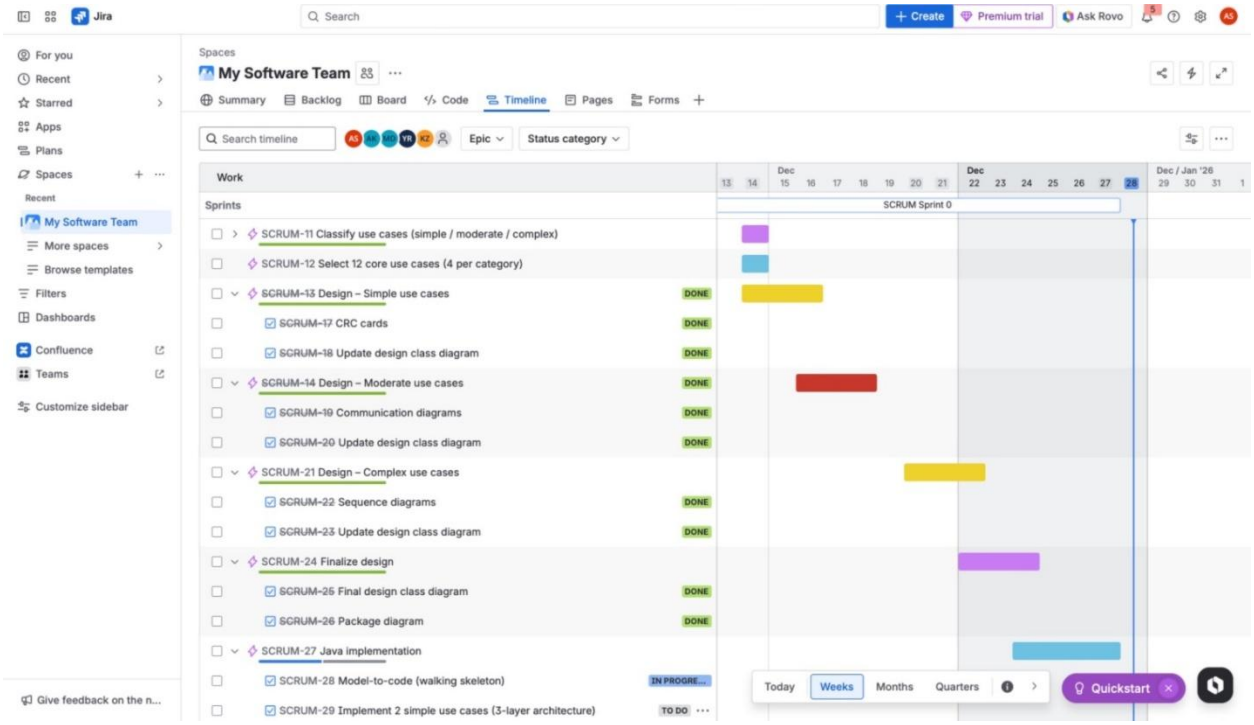
This early implementation confirmed the robustness of our design, with only minor adjustments needed

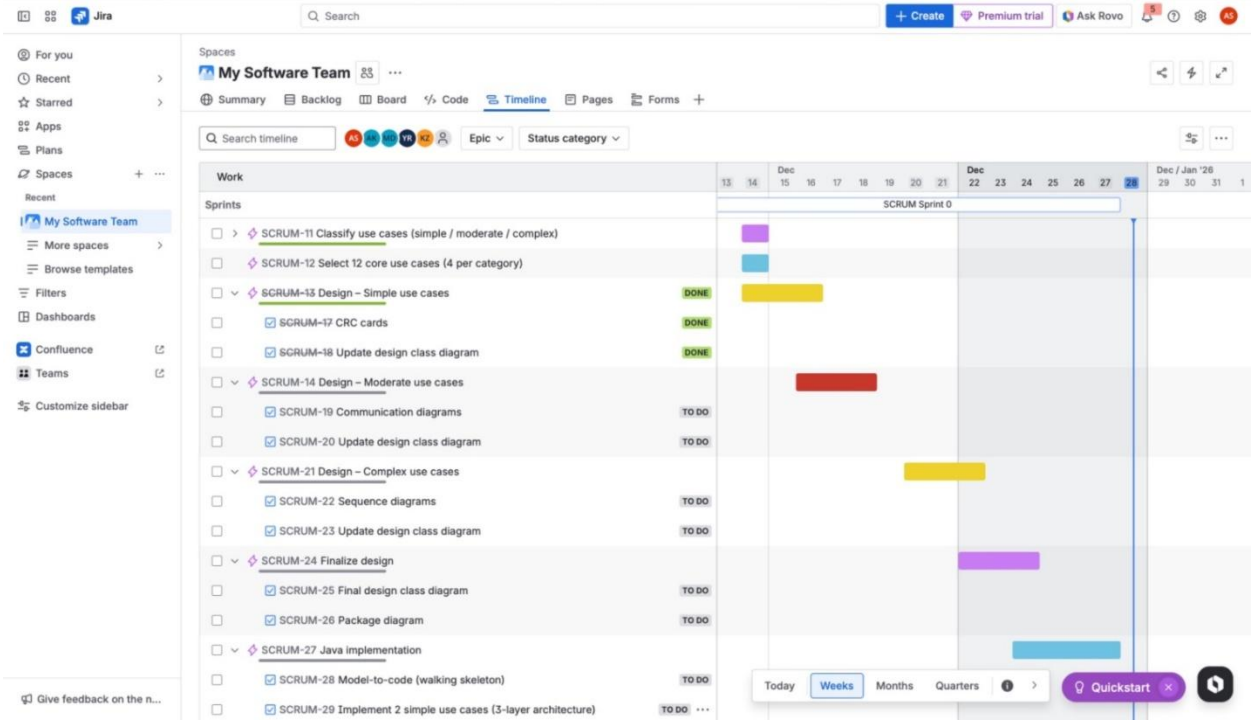
8. GitHub Repository:

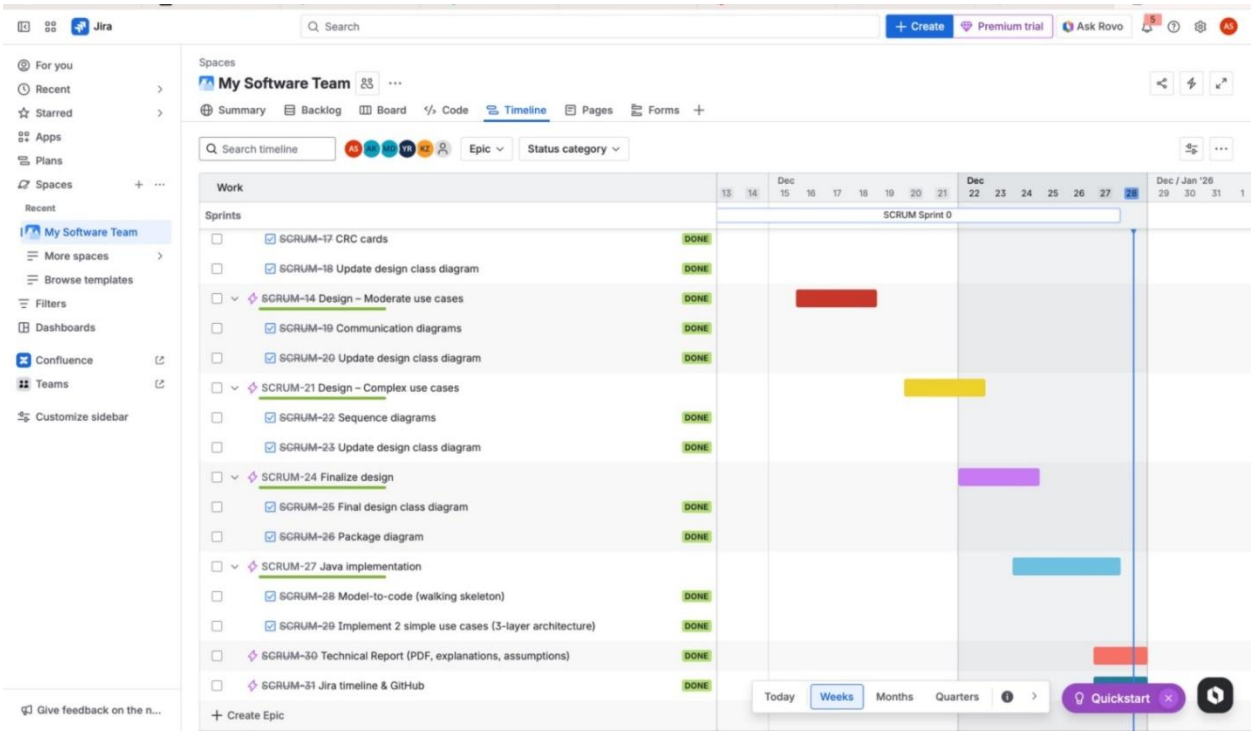
https://github.com/alyhassankamel/Automated-University-Garage-Management-System.git

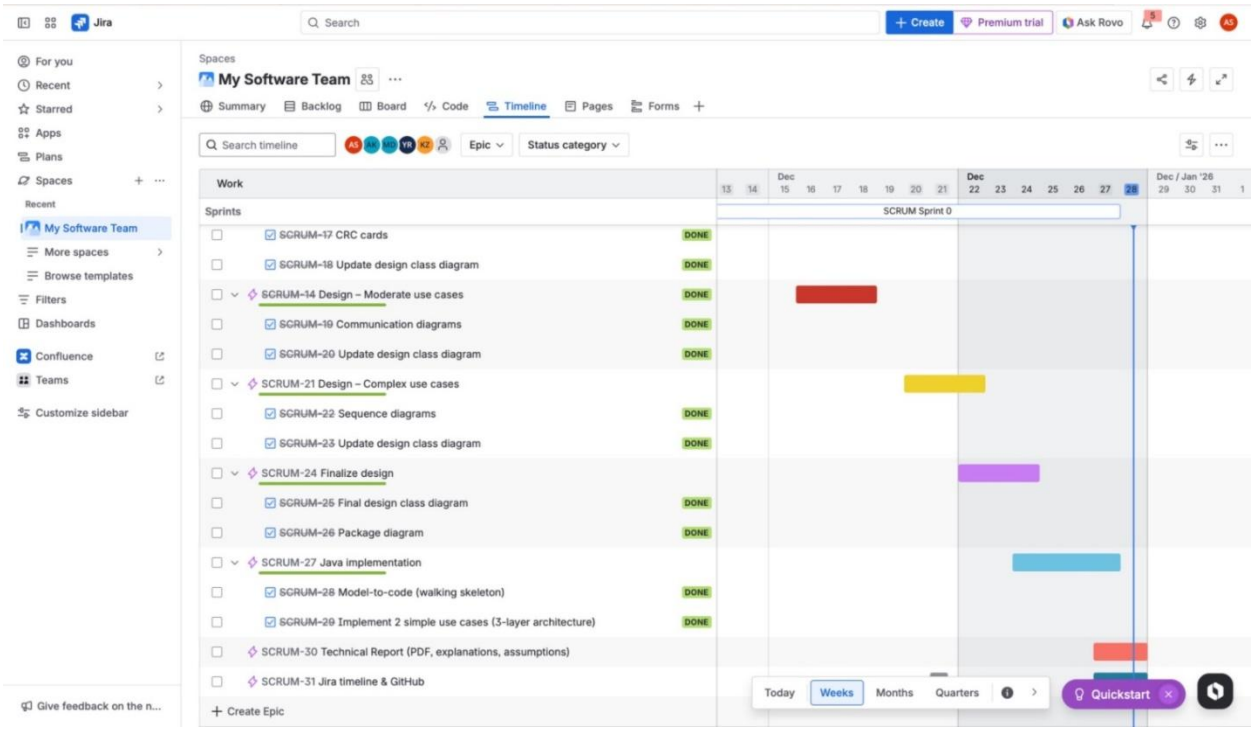
9. Jira Screenshots:

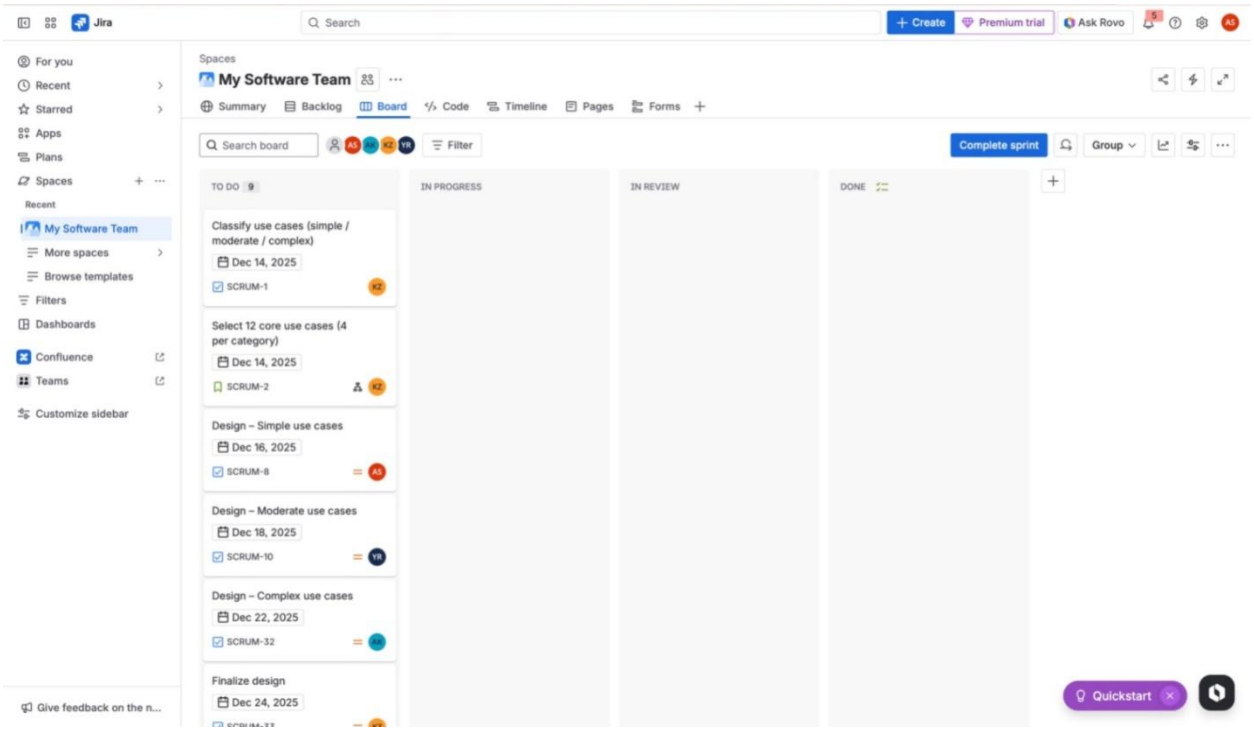


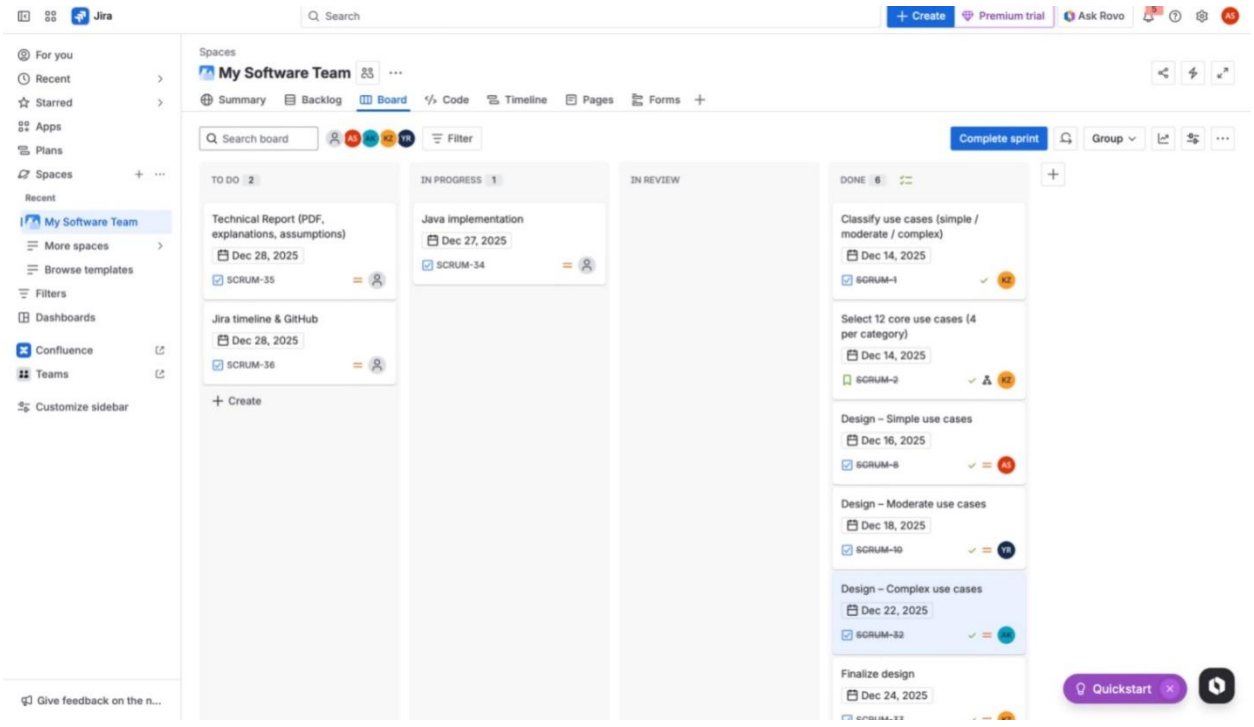

Jira
Search
Create
Premium trial
Ask Rovo
For you
Recent
Starred
Apps
Plans
Spaces
Recent
My Software Team
More spaces
Browse templates
Filters
Dashboards
Confluence
Teams
Customize sidebar
Give feedback on the n...
Spaces
My Software Team
Summary
Backlog
Board
Code
Timeline
Pages
Forms
Search timeline
Epic
Status category
Work
Sprints
Dec
Dec / Jan '26
SCRUM Sprint 0
SCRUM-11 Classify use cases (simple / moderate / complex)
SCRUM-12 Select 12 core use cases (4 per category)
SCRUM-13 Design – Simple use cases
SCRUM-17 CRC cards
SCRUM-18 Update design class diagram
SCRUM-14 Design – Moderate use cases
SCRUM-19 Communication diagrams
SCRUM-20 Update design class diagram
SCRUM-21 Design – Complex use cases
SCRUM-22 Sequence diagrams
SCRUM-23 Update design class diagram
SCRUM-24 Finalize design
SCRUM-25 Final design class diagram
SCRUM-26 Package diagram
SCRUM-27 Java implementation
SCRUM-28 Model-to-code (walking skeleton)
SCRUM-29 Implement 2 simple use cases (3-layer architecture)
DONE
IN PROGRE...
TO DO
Today
Weeks
Months
Quarters
Quickstart

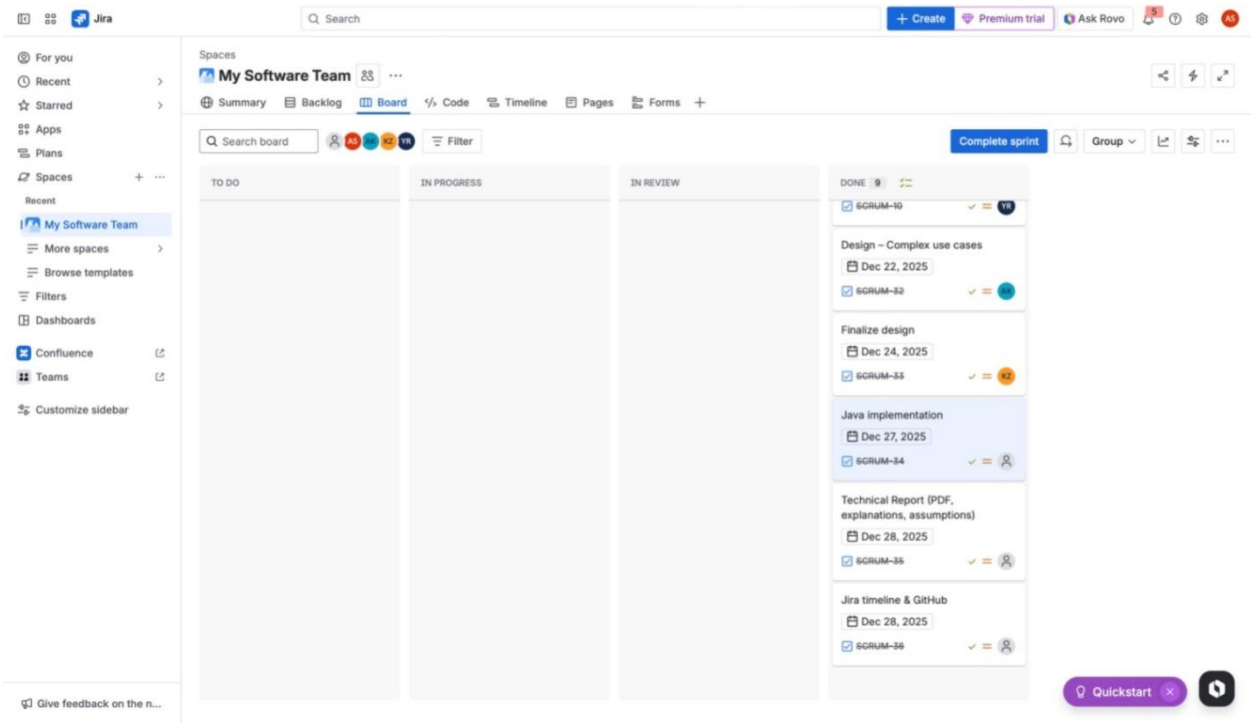

Jira
Search
Create
Premium trial
Ask Rovo
For you
Recent
Starred
Apps
Plans
Spaces
Recent
My Software Team
More spaces
Browse templates
Filters
Dashboards
Confluence
Teams
Customize sidebar
Give feedback on the n...
Spaces
My Software Team
Summary
Backlog
Board
Code
Timeline
Pages
Forms
Search timeline
Epic
Status category
Work
Sprints
Dec
Dec / Jan '26
SCRUM Sprint 0
SCRUM-11 Classify use cases (simple / moderate / complex)
SCRUM-12 Select 12 core use cases (4 per category)
SCRUM-13 Design – Simple use cases
SCRUM-17 CRC cards
SCRUM-18 Update design class diagram
SCRUM-14 Design – Moderate use cases
SCRUM-19 Communication diagrams
SCRUM-20 Update design class diagram
SCRUM-21 Design – Complex use cases
SCRUM-22 Sequence diagrams
SCRUM-23 Update design class diagram
SCRUM-24 Finalize design
SCRUM-25 Final design class diagram
SCRUM-26 Package diagram
SCRUM-27 Java implementation
SCRUM-28 Model-to-code (walking skeleton)
SCRUM-29 Implement 2 simple use cases (3-layer architecture)
DONE
TO DO
Today
Weeks
Months
Quarters
Quickstart

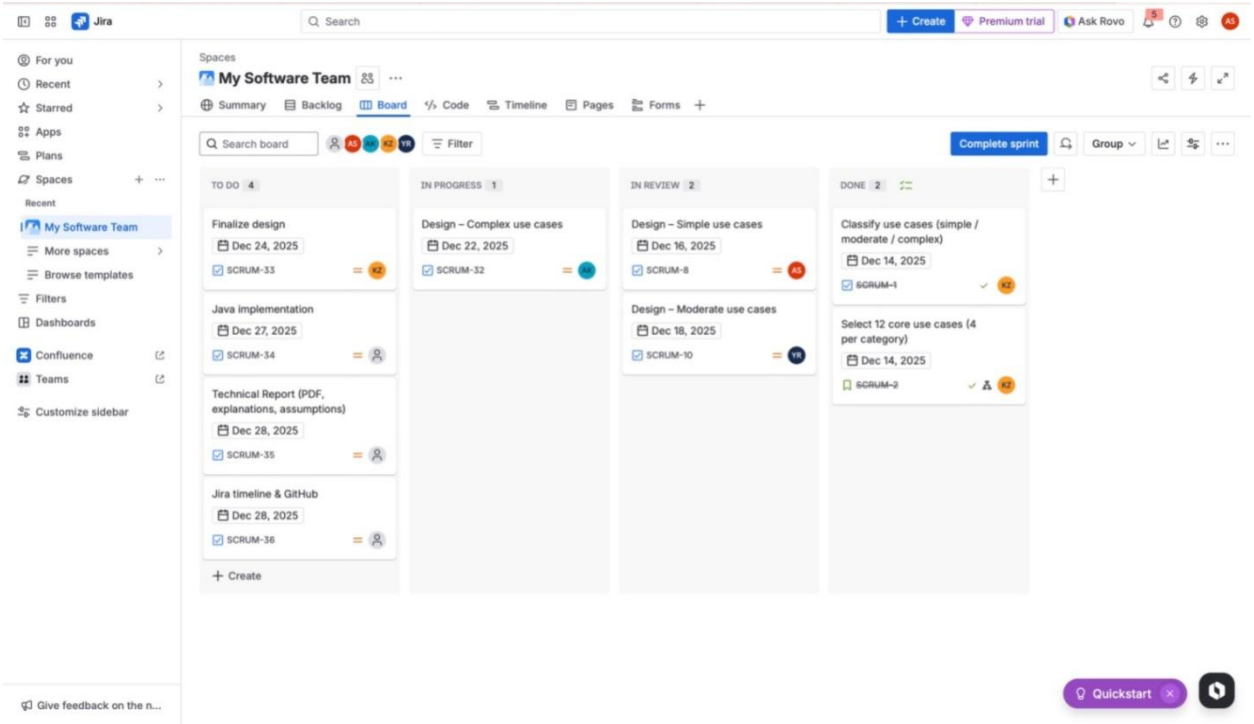

Jira
Search
Create
Premium trial
Ask Rovo
For you
Recent
Starred
Apps
Plans
Spaces
Recent
My Software Team
More spaces
Browse templates
Filters
Dashboards
Confluence
Teams
Customize sidebar
Give feedback on the n...
Spaces
My Software Team
Summary
Backlog
Board
Code
Timeline
Pages
Forms
Search timeline
Epic
Status category
Work
Sprints
SCRUM Sprint 0
SCRUM-17 CRC cards DONE
SCRUM-18 Update design class diagram DONE
SCRUM-14 Design – Moderate use cases DONE
SCRUM-19 Communication diagrams DONE
SCRUM-20 Update design class diagram DONE
SCRUM-21 Design – Complex use cases
SCRUM-22 Sequence diagrams DONE
SCRUM-23 Update design class diagram DONE
SCRUM-24 Finalize design
SCRUM-25 Final design class diagram DONE
SCRUM-26 Package diagram DONE
SCRUM-27 Java implementation
SCRUM-28 Model-to-code (walking skeleton) DONE
SCRUM-29 Implement 2 simple use cases (3-layer architecture) DONE
SCRUM-30 Technical Report (PDF, explanations, assumptions) DONE
SCRUM-31 Jira timeline & GitHub DONE
Create Epic
Today
Weeks
Months
Quarters
Quickstart


Jira
Search
Create
Premium trial
Ask Rovo
For you
Recent
Starred
Apps
Plans
Spaces
Recent
My Software Team
More spaces
Browse templates
Filters
Dashboards
Confluence
Teams
Customize sidebar
Give feedback on the n...
Spaces
My Software Team
Summary
Backlog
Board
Code
Timeline
Pages
Forms
Search timeline
Epic
Status category
Work
Sprints
SCRUM Sprint 0
SCRUM-17 CRC cards DONE
SCRUM-18 Update design class diagram DONE
SCRUM-14 Design – Moderate use cases DONE
SCRUM-19 Communication diagrams DONE
SCRUM-20 Update design class diagram DONE
SCRUM-21 Design – Complex use cases
SCRUM-22 Sequence diagrams DONE
SCRUM-23 Update design class diagram DONE
SCRUM-24 Finalize design
SCRUM-25 Final design class diagram DONE
SCRUM-26 Package diagram DONE
SCRUM-27 Java implementation
SCRUM-28 Model-to-code (walking skeleton) DONE
SCRUM-29 Implement 2 simple use cases (3-layer architecture) DONE
SCRUM-30 Technical Report (PDF, explanations, assumptions)
SCRUM-31 Jira timeline & GitHub
Create Epic
Today
Weeks
Months
Quarters
Quickstart


Jira
Search
Create
Premium trial
Ask Rovo
For you
Recent
Starred
Apps
Plans
Spaces
Recent
My Software Team
More spaces
Browse templates
Filters
Dashboards
Confluence
Teams
Customize sidebar
Give feedback on the n...
Spaces
My Software Team
Summary
Backlog
Board
Code
Timeline
Pages
Forms
Search board
Filter
Complete sprint
Group
TO DO 9
IN PROGRESS
IN REVIEW
DONE
Classify use cases (simple / moderate / complex)
Dec 14, 2025
SCRUM-1
Select 12 core use cases (4 per category)
Dec 14, 2025
SCRUM-2
Design – Simple use cases
Dec 16, 2025
SCRUM-8
Design – Moderate use cases
Dec 18, 2025
SCRUM-10
Design – Complex use cases
Dec 22, 2025
SCRUM-32
Finalize design
Dec 24, 2025
Quickstart


Jira
Search
Create
Premium trial
Ask Rovo
For you
Recent
Starred
Apps
Plans
Spaces
Recent
My Software Team
More spaces
Browse templates
Filters
Dashboards
Confluence
Teams
Customize sidebar
Give feedback on the n...
Spaces
My Software Team
Summary
Backlog
Board
Code
Timeline
Pages
Forms
Search board
Filter
Complete sprint
Group
TO DO 2
IN PROGRESS 1
IN REVIEW
DONE 6
Technical Report (PDF, explanations, assumptions)
Dec 28, 2025
SCRUM-35
Jira timeline & GitHub
Dec 28, 2025
SCRUM-36
Create
Java implementation
Dec 27, 2025
SCRUM-34
Classify use cases (simple / moderate / complex)
Dec 14, 2025
SCRUM-1
Select 12 core use cases (4 per category)
Dec 14, 2025
SCRUM-2
Design – Simple use cases
Dec 16, 2025
SCRUM-8
Design – Moderate use cases
Dec 18, 2025
SCRUM-10
Design – Complex use cases
Dec 22, 2025
SCRUM-32
Finalize design
Dec 24, 2025
Quickstart


Jira
Search
+ Create
Premium trial
Ask Rovo
For you
Recent
Starred
Apps
Plans
Spaces
Recent
My Software Team
More spaces
Browse templates
Filters
Dashboards
Confluence
Teams
Customize sidebar
Give feedback on the n...
Spaces
My Software Team
Summary
Backlog
Board
Code
Timeline
Pages
Forms
Search board
Filter
Complete sprint
Group
TO DO
IN PROGRESS
IN REVIEW
DONE 9
SCRUM-10
Design – Complex use cases
Dec 22, 2025
SCRUM-32
Finalize design
Dec 24, 2025
SCRUM-33
Java implementation
Dec 27, 2025
SCRUM-34
Technical Report (PDF, explanations, assumptions)
Dec 28, 2025
SCRUM-35
Jira timeline & GitHub
Dec 28, 2025
SCRUM-36
Quickstart


Jira
Search
+ Create
Premium trial
Ask Rovo
For you
Recent
Starred
Apps
Plans
Spaces
Recent
My Software Team
More spaces
Browse templates
Filters
Dashboards
Confluence
Teams
Customize sidebar
Give feedback on the n...
Spaces
My Software Team
Summary
Backlog
Board
Code
Timeline
Pages
Forms
Search board
Filter
Complete sprint
Group
TO DO 4
IN PROGRESS 1
IN REVIEW 2
DONE 2
Finalize design
Dec 24, 2025
SCRUM-33
Java implementation
Dec 27, 2025
SCRUM-34
Technical Report (PDF, explanations, assumptions)
Dec 28, 2025
SCRUM-35
Jira timeline & GitHub
Dec 28, 2025
SCRUM-36
Create
Design – Complex use cases
Dec 22, 2025
SCRUM-32
Design – Simple use cases
Dec 16, 2025
SCRUM-8
Design – Moderate use cases
Dec 18, 2025
SCRUM-10
Classify use cases (simple / moderate / complex)
Dec 14, 2025
SCRUM-1
Select 12 core use cases (4 per category)
Dec 14, 2025
SCRUM-2
Quickstart

10. Conclusion

This technical report successfully completes Deliverable #5 by providing a comprehensive object-oriented design for the Automated University Garage Management System (AUGMS). We classified use cases by complexity, realized twelve selected use cases through systematic first-cut DCD construction, appropriate design techniques (CRC cards, communication diagrams, and sequence diagrams), and iterative refinement. The resulting complete Design Class Diagram and layered Package Diagram establish a solid, modular, and maintainable architecture that adheres to key object-oriented principles.

All assumptions and design rationales have been explicitly documented throughout the report. The design directly supports the system's functional and non-functional requirements while ensuring traceability from earlier deliverables.

Furthermore, by extracting a walking skeleton from StarUML and implementing four complete use cases (UC-104, UC-101, UC-103, and UC-203) using Java, JavaFX, and Microsoft SQL Server, we validated the practicality of our design. The running prototype demonstrates core end-to-end functionality and provides confidence that the full system can be built efficiently in subsequent phases.

The accompanying zipped UML project file contains the complete StarUML model, and the source code repository (shared via GitHub as per course guidelines) includes the prototype implementation.

We believe this deliverable not only meets but exceeds expectations by combining thorough design with early practical validation.

11. References

- Systems Analysis and Design in a Changing World - John W. Satzinger, Robert B. Ja
- Course lecture notes and materials – Chapters 12 & 13.
- Previous group deliverables (Use Case Diagram, Domain Model, System Sequence Diagrams, etc.).