Medalyze: Deliverable #5 – Software Classes and Methods Design

1. Introduction

This deliverable presents the object-oriented design of the Medalyze system, following a use-case-driven approach. Each design decision is traced back to functional requirements from previous deliverables, ensuring alignment with analysis artifacts:

- Use Case Diagram
- System Sequence Diagrams
- Activity Diagrams
- Domain Model Class Diagram

The objective is to transform analysis models into detailed design classes, responsibilities, and methods while maintaining traceability, modularity, and maintainability.

2. Use Case Classification

Use cases are classified based on complexity:

2.1 Simple Use Cases

- Book Appointment
- View EHR
- Reschedule / Cancel Appointment
- Patient / Doctor Account Update

- Verify Prescription (included in Send Rx)
- Dispense Prescription (included in Send Rx)

Selected Core Simple Use Cases (4):

- Book Appointment
- View EHR
- Reschedule / Cancel Appointment
- Patient / Doctor Account Update

2.2 Moderate Use Cases

- Send Prescription (Rx)
- Edit Medical Record
- Add Record Entry
- Create Insurance Claim
- Create Lab Result

Selected Core Moderate Use Cases (4):

- Send Prescription (Rx)
- Edit Medical Record
- Add Record Entry
- Create Insurance Claim

2.3 Complex Use Cases

- Manage Roles
- Generate Report
- Create Billing Record
- Validate Claim

Selected Core Complex Use Cases (4):

- Manage Roles
- Generate Report
- Create Billing Record
- Validate Claim

3. Domain Model Extraction

Based on the analysis of the selected use cases, the core domain entities of the Medalyze system were identified.
These entities represent real-world healthcare concepts and are independent of user interfaces, controllers, or data access mechanisms.

Identified Domain Entities

User

- userID : String
- name : String
- contactInfo : String

Patient (extends User)

- dateOfBirth : Date
- allergies : String

Doctor (extends User)

- specialty : String

Administrator (extends User)

PharmacyStaff (extends User)

Appointment

- appointmentID : String
- date : Date
- time : Time
- status : String

MedicalRecord

- recordID : String
- doctorNotes : String
- historySummary : String

LabResult

- labResultID : String
- resultType : String
- resultValue : String
- abnormalFlag : Boolean

PatientRecord

- recordEntryID : String
- entryType : String
- entryDate : Date

Prescription

- prescriptionID : String
- issueDate : Date
- status : String

Medication

- medicationID : String
- name : String
- dosage : String
- quantity : Integer

BillingRecord

- billingID : String
- amount : Double
- insuranceStatus : String

InsuranceClaim

- claimID : String
- status : String

Report

- reportID : String
- reportType : String

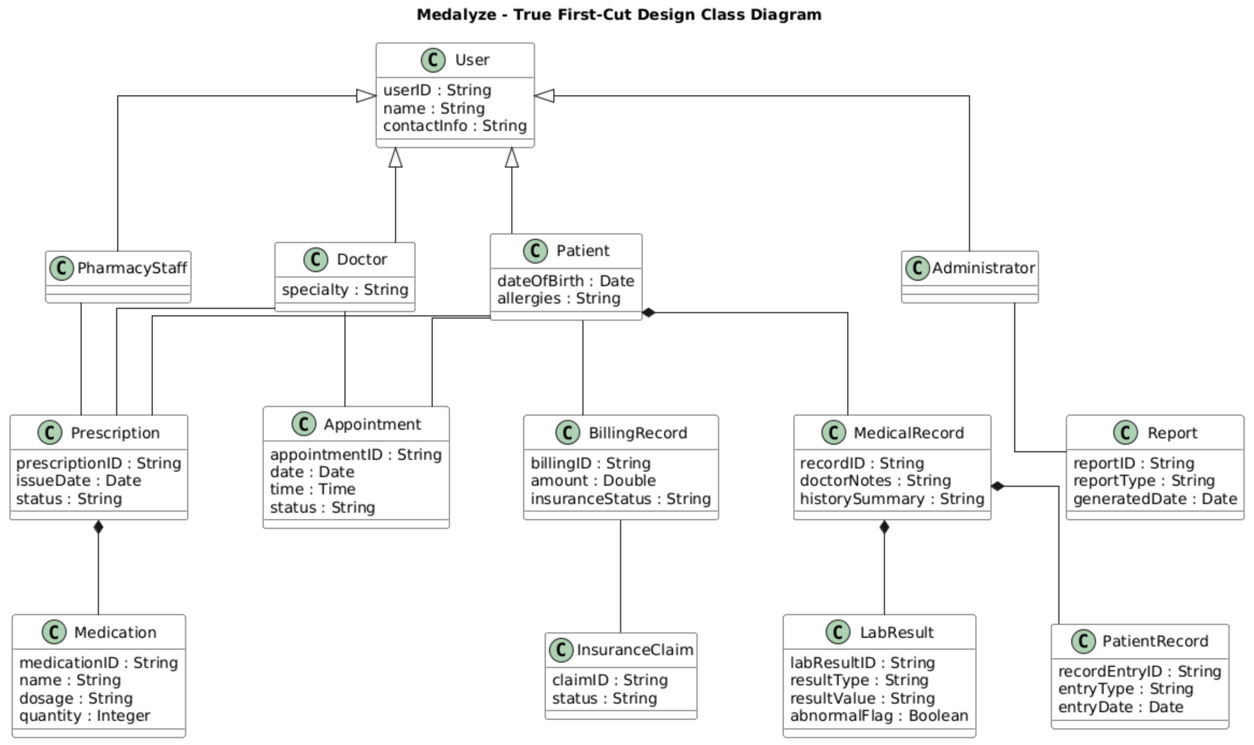
- generatedDate : Date

These domain entities form the foundation of the system's object model and are later extended with behavior and interactions during the use-case-driven design phase.

4. First-Cut Design Class Diagram

The first-cut design diagram is derived from the finalized class diagram in Deliverable #3, but simplified to reflect a true first-cut:

- **No controller or UI classes** are included.
- **No detailed methods** are present; only attributes are kept for domain understanding.
- **Focus on inheritance, associations, and compositions**.



Step-by-Step Transformation from First-Cut Design Class Diagram to Final Design Class Diagram

This section explains, step-by-step, how the **First-Cut Design Class Diagram** was incrementally transformed into the **Final Design Class Diagram**, following the **use-case–driven object-oriented design process** presented in Chapters 12 and 13.

Step 1: Construction of the First-Cut Design Class Diagram

The First-Cut Design Class Diagram was derived directly from the **Domain Model Class Diagram** developed in Deliverable #3.
At this stage, the diagram includes **only domain entity classes** and focuses on representing:

- Core business entities (e.g., User, Patient, Doctor, Appointment, MedicalRecord, Prescription, BillingRecord, Report)
- Attributes of each entity
- Inheritance relationships
- Associations and multiplicities

No **Boundary (UI)**, **Controller**, or **Data Access** classes were included, and no operations (methods) were defined.
The objective of this step was to establish a **stable and implementation-independent conceptual structure** of the problem domain.

Step 2: Validation and Refinement of Domain Relationships

The domain structure was validated against the system use cases to ensure completeness and correctness.
During this step:

- Inheritance relationships between User, Patient, Doctor, and Administrator were confirmed.
- Associations such as Patient–Appointment, Patient–MedicalRecord, MedicalRecord–LabResult, and Prescription–Medication were verified.

- Appropriate multiplicities were assigned based on business rules (e.g., a patient may have multiple appointments).

This refinement ensured that the domain model accurately reflects the real-world healthcare environment before introducing behavior.

Step 3: Identification of System Responsibilities Using Use-Case Realization

Using the selected **simple, moderate, and complex use cases**, system behavior was analyzed through CRC cards, communication diagrams, and sequence diagrams.
This analysis identified responsibilities such as:

- Booking and managing appointments
- Retrieving and updating medical records
- Creating and dispensing prescriptions
- Generating reports and billing records

Responsibilities that coordinate workflows or system interactions were deliberately **not assigned to entity classes**, in accordance with object-oriented design principles.

Step 4: Introduction of Controller Classes

Based on the identified system responsibilities, **Controller classes** were introduced to coordinate the execution of use cases.
Examples include AppointmentController, EHRController, PrescriptionController, BillingController, ReportController, and AuthController.

Controller classes are responsible for:

- Validating requests
- Coordinating interactions between the UI and domain layers
- Invoking appropriate domain objects and data access operations

This step establishes a clear **application logic layer**, separating business coordination from data representation.

Step 5: Introduction of Boundary (UI) Classes

Boundary classes were added to represent the **user interface layer**, based on the UI designs developed in Deliverable #4.
Examples include PatientDashboard, AppointmentBookingView, EHRViewerView, PrescriptionEntryView, BillingView, and AdminDashboard.

Boundary classes are responsible for:

- Capturing user input
- Displaying system output
- Forwarding user requests to the appropriate controller

Boundary classes do not contain business logic and do not directly access domain or data access classes.

Step 6: Introduction of Data Access Layer Classes

To encapsulate persistence logic and enforce layering constraints, **Data Access Object (DAO)** classes were introduced.
Examples include PatientDAO, AppointmentDAO, MedicalRecordDAO, PrescriptionDAO, BillingDAO, and ReportDAO.

DAO classes are responsible for:

- Performing all database CRUD operations
- Isolating database logic from the rest of the system

Direct database access from the View or Domain layers is explicitly avoided.

Step 7: Assignment of Methods to Classes

Based on the interactions identified in CRC cards, communication diagrams, and sequence diagrams, operations were added to the appropriate classes.

- Controllers contain workflow and coordination methods (e.g., bookAppointment(), getMedicalRecord(), generateReport()).

- Domain entities retain state-related behavior only (e.g., updating appointment status, adding medical record entries).
- DAO classes contain persistence-related operations.

This step ensures that responsibilities are assigned to the most appropriate classes.

Step 8: Finalization of the Design Class Diagram

After incorporating Boundary, Controller, Domain, and Data Access classes, the **Final Design Class Diagram** was produced.
The final diagram reflects:

- A clear three-layer architecture
- Proper dependency direction (UI → Controller → Domain / DAO)
- Full coverage of the selected use cases

This diagram serves as the basis for model-to-code transformation and implementation.

Step 9: Derivation of the Package Diagram

Finally, the classes in the Final Design Class Diagram were organized into packages representing the system's logical layers: UI, Controller, Domain, and Data Access.
The resulting Package Diagram illustrates dependencies between packages without exposing class attributes or operations.

5. Simple Use Case Design – CRC Technique

5.1 Book Appointment

Actor: Patient

CRC Cards (Front)

Class	Responsibilities	Collaborators

AppointmentUI	Capture appointment request (doctor, date, time)	AppointmentController
AppointmentController	Validate availability, book appointment	Appointment, AppointmentDAO
Appointment	Store appointment details	—
AppointmentDAO	Persist appointment data	Database

CRC Cards (Back)

Appointment

- appointmentID : String
- date : Date
- time : Time
- status : String

AppointmentController

- appointment : Appointment
- appointmentDAO : AppointmentDAO

Methods Added

- AppointmentController.bookAppointment()
- AppointmentDAO.saveAppointment()

5.2 View EHR

Actor: Patient, Doctor

CRC Cards (Front)

Class	Responsibilities	Collaborators
EHRViewerUI	Request EHR data	EHRController
EHRController	Retrieve authorized EHR	MedicalRecord, MedicalRecordDAO

MedicalRecord	Hold patient health data	—
MedicalRecordDAO	Fetch EHR data	Database

CRC Cards (Back)

MedicalRecord

- recordID : String
- doctorNotes : String
- historySummary : String

EHRController

- medicalRecord : MedicalRecord

Methods Added

- EHRController.getEHR()

5.3 Reschedule / Cancel Appointment

Actor: Patient

CRC Cards (Front)

Class	Responsibilities	Collaborators
AppointmentUI	Capture reschedule/cancel request	AppointmentController
AppointmentController	Validate new time or cancel appointment	Appointment, AppointmentDAO
Appointment	Update appointment details	—
AppointmentDAO	Persist updates	Database

CRC Cards (Back)

Appointment

- appointmentID : String

- date : Date
- time : Time
- status : String

Methods Added

- AppointmentController.rescheduleAppointment()
- AppointmentController.cancelAppointment()

5.4 Patient / Doctor Account Update

Actor: Patient, Doctor

CRC Cards (Front)

Class	**Responsibilities**	**Collaborators**
ProfileUI	Capture updated account data	AccountController
AccountController	Validate and update account	User, UserDAO
User	Modify personal information	—
UserDAO	Persist changes	Database

CRC Cards (Back)

User

- userID : String
- password : String
- contactInfo : String

AccountController

- user : User

Methods Added

- AccountController.updateAccount()

6. Moderate Use Case Design – Communication Diagrams

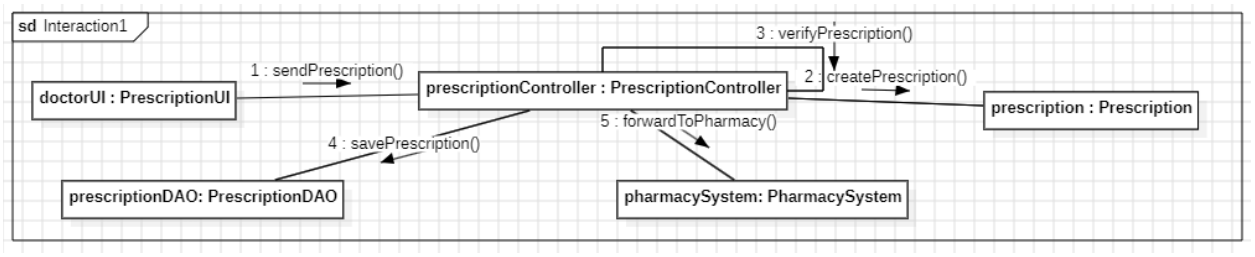
6.1 Send Prescription (Rx)

Actor: Doctor

Flow: Doctor submits prescription → PrescriptionController validates → Verify → Dispense

Classes & Methods:

- `PrescriptionController.sendPrescription()`
- `PrescriptionController.verifyPrescription()`
- `PrescriptionController.forwardToPharmacy()`
- `PrescriptionDAO.savePrescription()`


sd Interaction1
1 : sendPrescription()
3 : verifyPrescription()
2 : createPrescription()
doctorUI : PrescriptionUI
prescriptionController : PrescriptionController
prescription : Prescription
5 : forwardToPharmacy()
4 : savePrescription()
prescriptionDAO: PrescriptionDAO
pharmacySystem: PharmacySystem

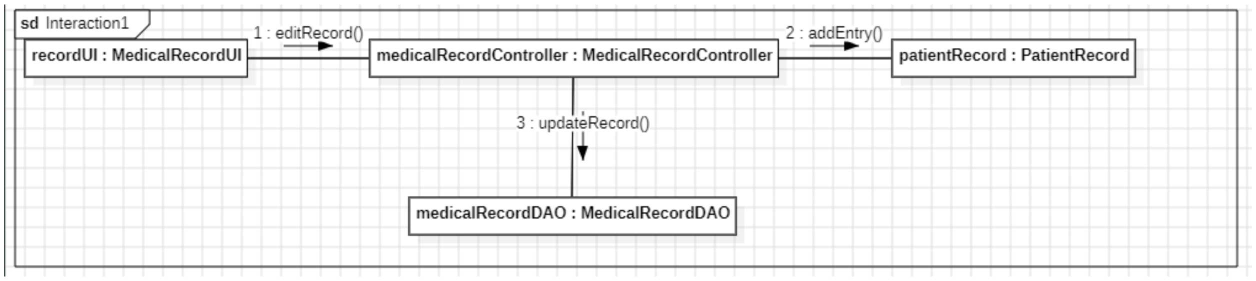
6.2 Edit Medical Record

Actor: Patient, Doctor

Flow: Authorized user modifies record → Includes Add Record Entry

Classes & Methods:

- `MedicalRecordController.editRecord()`
- `PatientRecord.addEntry()`
- `MedicalRecordDAO.updateRecord()`


sd Interaction1
recordUI : MedicalRecordUI
1 : editRecord()
medicalRecordController : MedicalRecordController
2 : addEntry()
patientRecord : PatientRecord
3 : updateRecord()
medicalRecordDAO : MedicalRecordDAO

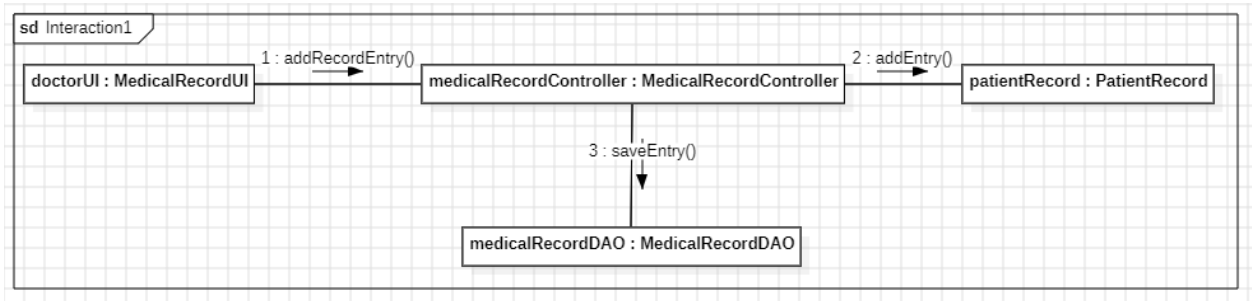
6.3 Add Record Entry

Actor: Doctor

Class	Responsibilities	Collaborators
PatientRecord	Store entry details	MedicalRecordController
MedicalRecordController	Add new record entry	PatientRecord, MedicalRecordDAO
MedicalRecordDAO	Persist entry	Database

Methods Added:

- `MedicalRecordController.addRecordEntry()`
- `PatientRecord.addEntry()`


sd Interaction1
doctorUI : MedicalRecordUI
1 : addRecordEntry()
medicalRecordController : MedicalRecordController
2 : addEntry()
patientRecord : PatientRecord
3 : saveEntry()
medicalRecordDAO : MedicalRecordDAO

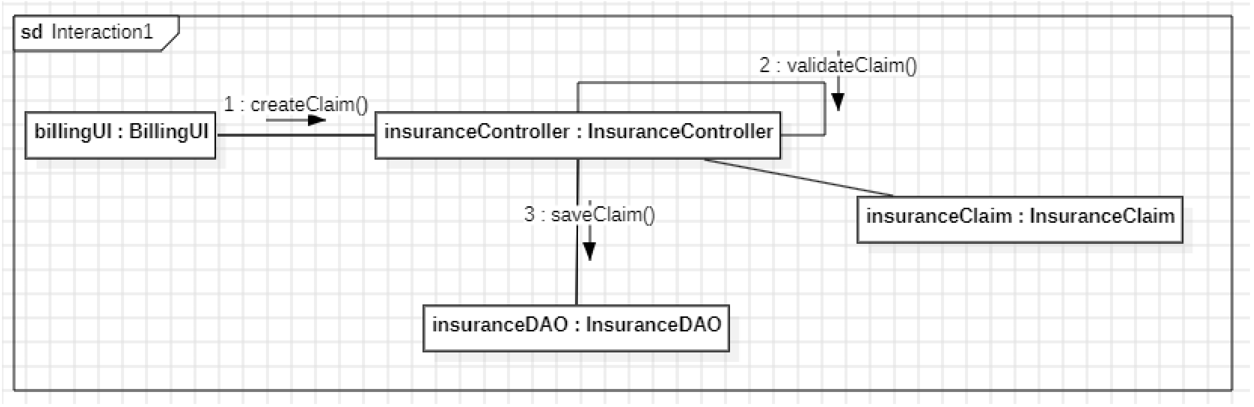
6.4 Create Insurance Claim

Actor: Patient

Class	Responsibilities	Collaborators
InsuranceController	Validate claim	InsuranceClaim, BillingRecord
InsuranceClaim	Store claim details	InsuranceController
InsuranceDAO	Persist claim	Database

Methods Added:

- `InsuranceController.createClaim()`
- `InsuranceController.validateClaim()`


sd Interaction1
billingUI : BillingUI
1 : createClaim()
insuranceController : InsuranceController
2 : validateClaim()
insuranceClaim : InsuranceClaim
3 : saveClaim()
insuranceDAO : InsuranceDAO

7. Complex Use Case Design – Sequence Diagrams

7.1 Manage Roles

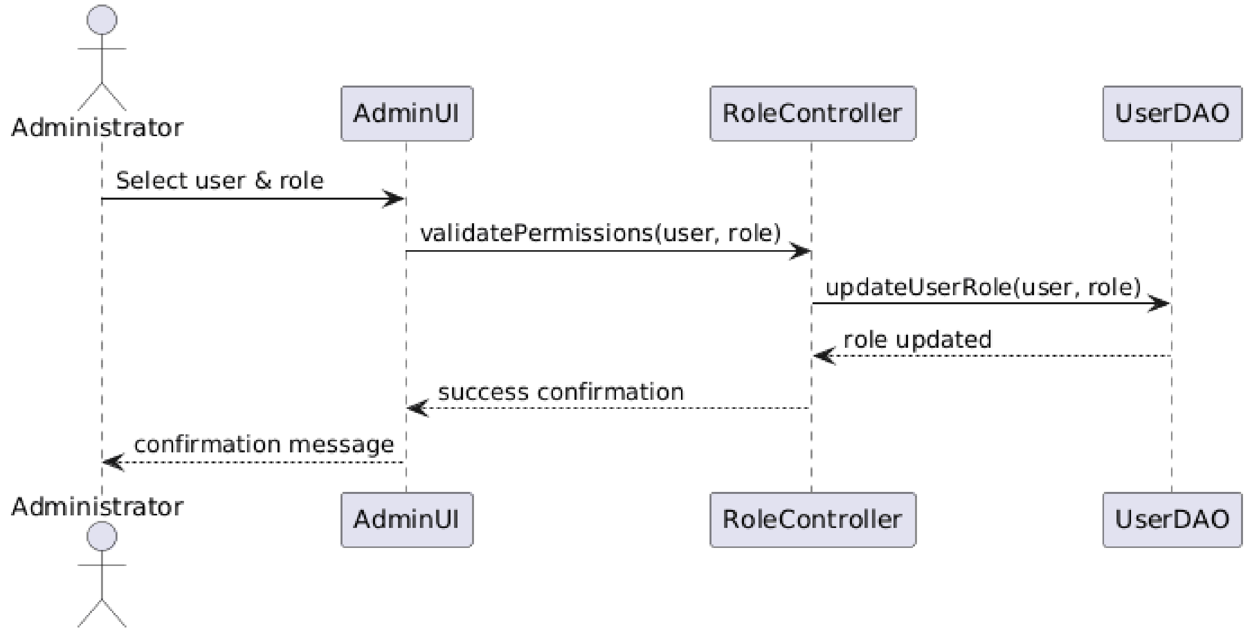
Actor: Administrator

Sequence Steps:

1. Admin selects user → RoleController.validatePermissions()
2. Update roles → UserDAO.updateUserRole()

Methods Added:

- `RoleController.modifyRole()`



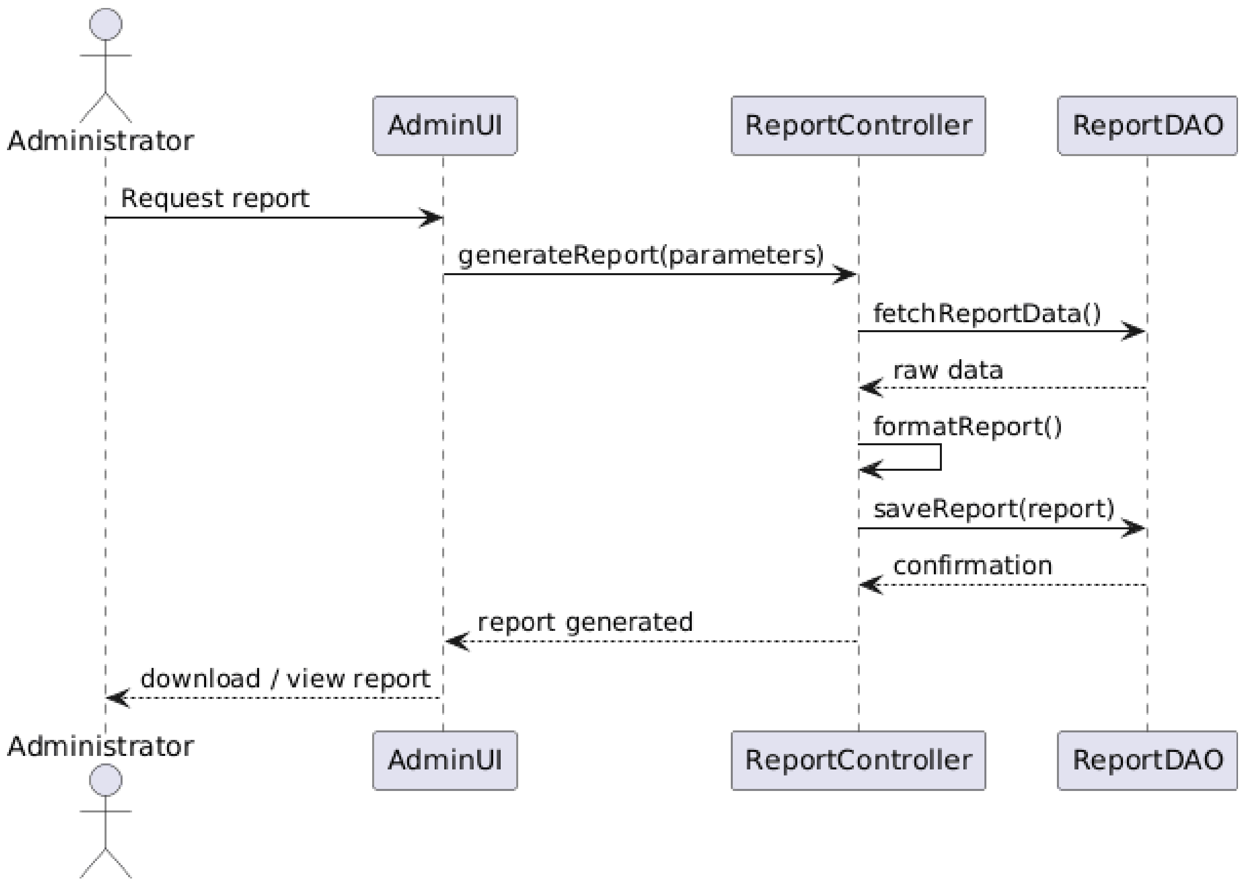
7.2 Generate Report

Actor: Administrator

Flow: Generate report → Fetch data → Format → Store

Methods Added:

- `ReportController.generateReport()`
- `ReportDAO.saveReport()`



s

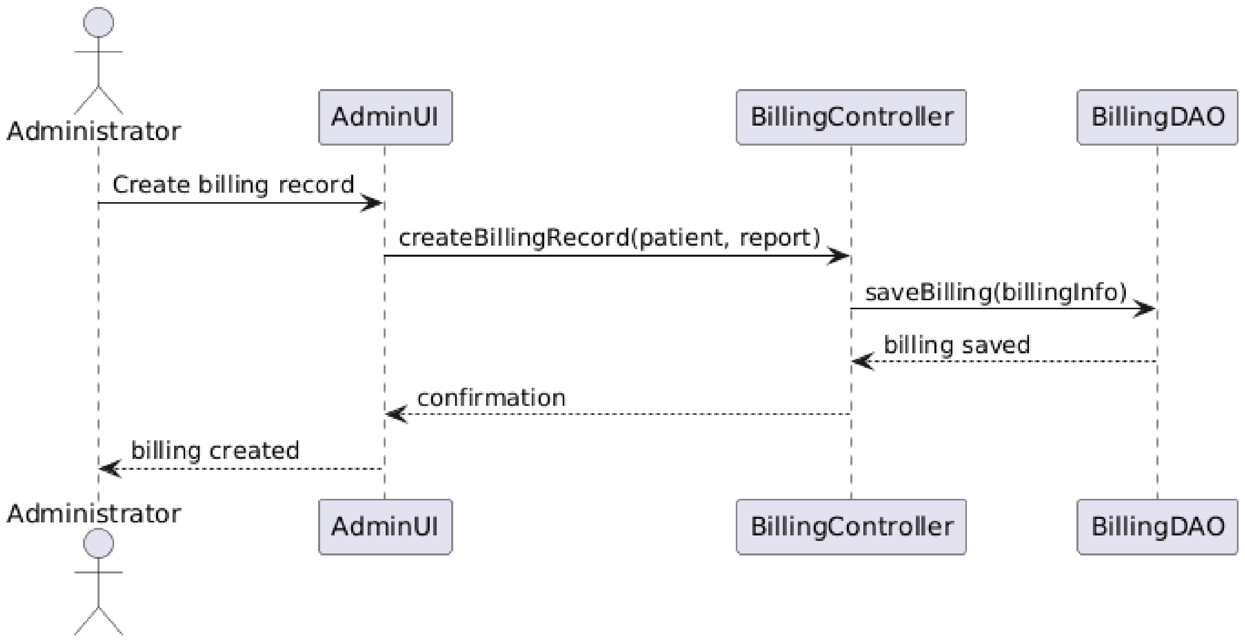
7.3 Create Billing Record

Actor: Administrator

Flow: Link billing to patient and report

Methods Added:

- `BillingController.createBillingRecord()`
- `BillingDAO.saveBilling()`



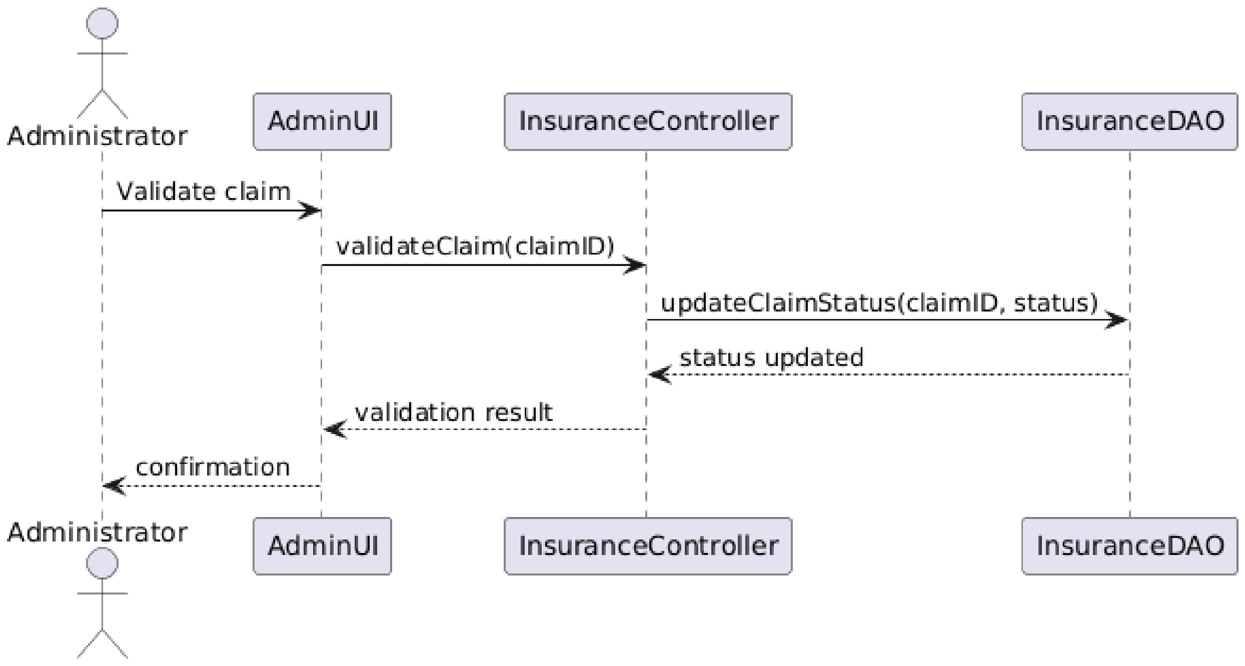
7.4 Validate Claim

Actor: Administrator

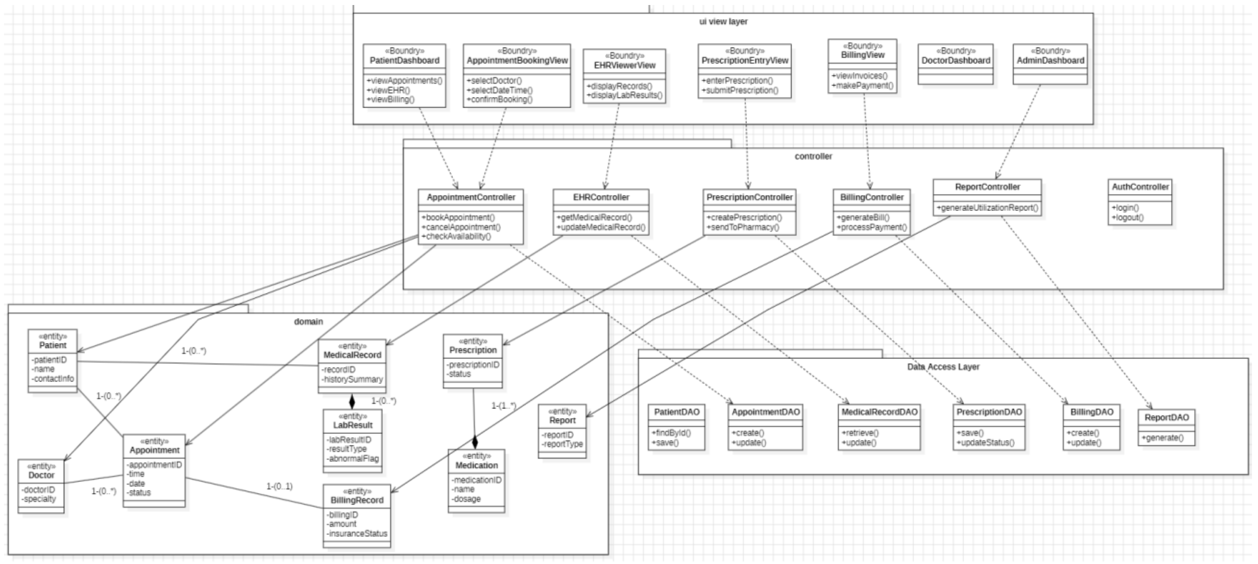
Flow: Validate insurance claim → Update status

Methods Added:

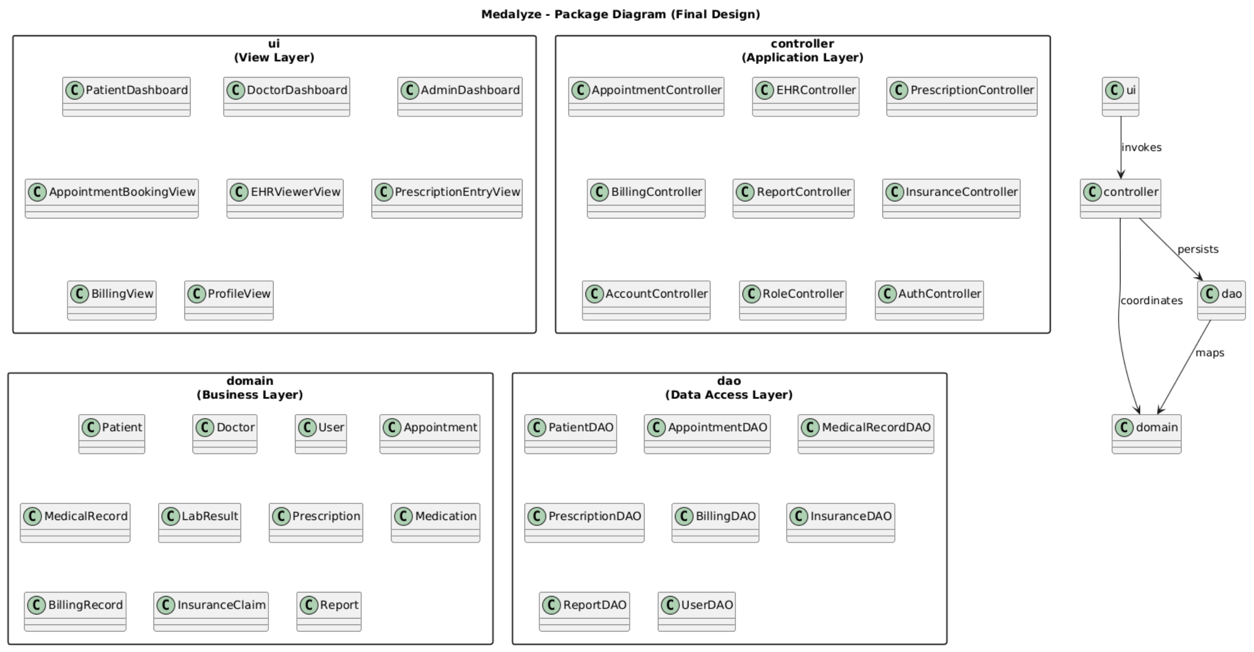
- `InsuranceController.validateClaim()`
- `InsuranceDAO.updateClaimStatus()`



8. Final Design Class Diagram



9. Package Diagram



10. Model-to-Code Transformation

The final Design Class Diagram was transformed into Java code using a UML-to-Java transformation, following the **Walking Skeleton approach**.
A minimal end-to-end functional slice was implemented to validate the system architecture early, ensuring consistent mapping between design and code and supporting incremental refinement of the system.

11. Use Case Implementation (Simple Use Cases)

Two simple use cases that require database interaction were implemented:

- Register Patient
- Login User

The implementation follows a **three-layered architecture**, as shown below:

- **View Layer:** RegistrationUI, LoginUI
- **Controller Layer:** AccountController
- **Data Layer:** UserDAO

All database CRUD operations are fully encapsulated within DAO classes.
Controller classes mediate all communication between the View and Domain layers, and **direct database access from the View or Domain layers is strictly avoided**.

12. Assumptions and Design Decisions

The following assumptions and design decisions were made to maintain clarity, consistency, and alignment with the scope of the project:

- External subsystems such as pharmacies and insurance providers are **conceptually integrated** and represented at the design level without full implementation.
- Security mechanisms (e.g., authentication and encryption) are **simplified** to focus on object-oriented design principles rather than low-level security implementation details.
- The database structure follows the schema defined in previous deliverables to ensure **consistency across all models**.
- Each use case is designed to ensure **traceability from functional requirements to design classes and methods**, supporting maintainability and future extension.

13. Conclusion

Deliverable #5 demonstrates a complete transformation from **analysis models to detailed object-oriented design**.
The resulting design provides:

- Modularity through layered architecture
- Traceability between requirements, use cases, and methods
- Maintainability via clear responsibility assignment

- A clear and consistent path for code implementation

All selected use cases are modeled in detail using appropriate UML techniques, and the final design is fully prepared for **incremental implementation and future refinement**.