University Equipment Loan System (UELoS)

SWEN-383 — Software Design Principles | Option B: Build a Small Application

Author: Erion Vrella, Daorsa Salihu, Vlera Merovci | Date: October 25, 2025

1. Project Description & Scope

The **University Equipment Loan System (UELoS)** is a small JavaScript web application that allows university students to borrow and return campus equipment such as laptops, cameras, and lab tools.

The system tracks which student has borrowed what, manages due dates, calculates fines for late returns, and allows staff members to approve requests, manage inventory, and oversee reports.

The goal is to demonstrate **clean, maintainable design** following **SOLID** and **GRASP** principles.

In Scope

In Scope	Out of Scope
• **Users & Roles: Student, Staff (combined role for administrative and approval duties)** • **Auth: Registration, login, session management, and password reset** • **Catalog: Create/edit equipment, categories, and availability** • **Loan Lifecycle: Request → Approve/Reject → Collect → Return → Close** • **Overdue Handling: Scheduled checks, status updates, and notifications** • **Fines: Policy-based fine calculation (per day / grace period), online payment (card)** • **Payments: Pay overdue fines, generate receipts, process refunds** • **Reports: Equipment status, active loans, overdue summaries, and total payments collected** • **Audit Log: Record key actions such as approvals, returns, and payments** • **Non-Functional: Input validation, error handling, data persistence, basic security, and logging**	• External campus systems (LDAP/SSO), multi-campus logistics, or advanced BI dashboards • Dedicated mobile applications (future integration possible via APIs) • Machine learning–based predictions or demand analytics

2. System Actors & Main Use Cases

Actors

- **Student:** Requests to borrow equipment, returns items, and pays fines if overdue.
- **Staff Member:** Manages catalog, reviews and approves/rejects loan requests, oversees reports, and manages fines/payments.
- *Note: Administrative functions (inventory, reporting, payments) are unified within the Staff role to simplify the system while keeping logical separation between approval and management tasks.*

Main Use Cases

- Student registers and logs in
- Student browses available equipment and submits a loan request
- Staff reviews the request and approves or rejects it
- Student collects equipment and loan starts
- Student returns equipment and system closes the loan
- Scheduler checks for overdue loans and notifies users
- System computes fines and allows students to pay online
- Staff manages catalog and monitors reports

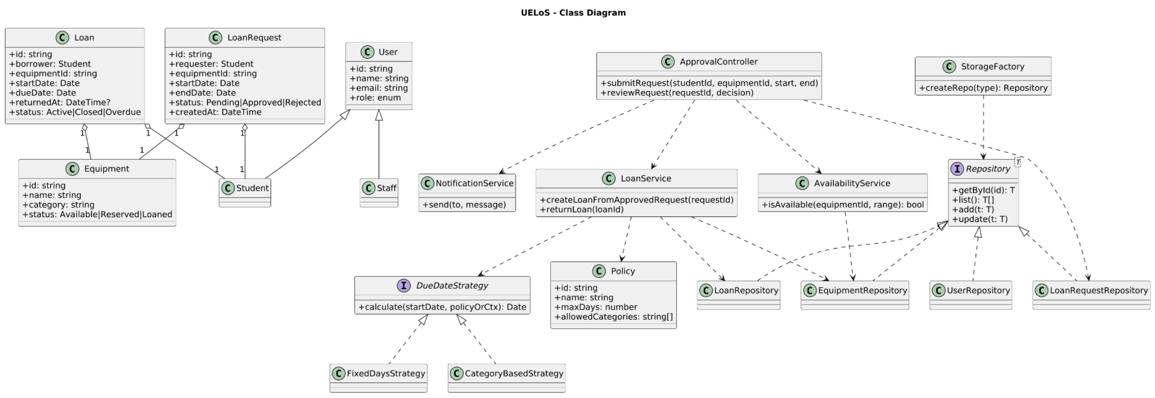
3. UML Design Drafts

3.1 Class Diagram (overview)

- **User(id, name, email, role)** → *Student*, *Staff* (inheritance)
- **Equipment(id, name, category, status)**
- **LoanRequest(id, student, equipment, dates, status)**
- **Loan(id, student, equipment, startDate, dueDate, returnedAt, status)**
- **Fine(id, loan, amount, status, paidAt)**
- **Repository Interface:** `UserRepo`, `EquipmentRepo`, `LoanRequestRepo`, `LoanRepo`, `FineRepo`
- **ApprovalController** — handles submission and review workflows
- **LoanService** — manages creation, return, and overdue logic
- **AvailabilityService** — checks and updates equipment status
- **PaymentService** — processes online fine payments
- **ReportService** — aggregates system-wide summaries

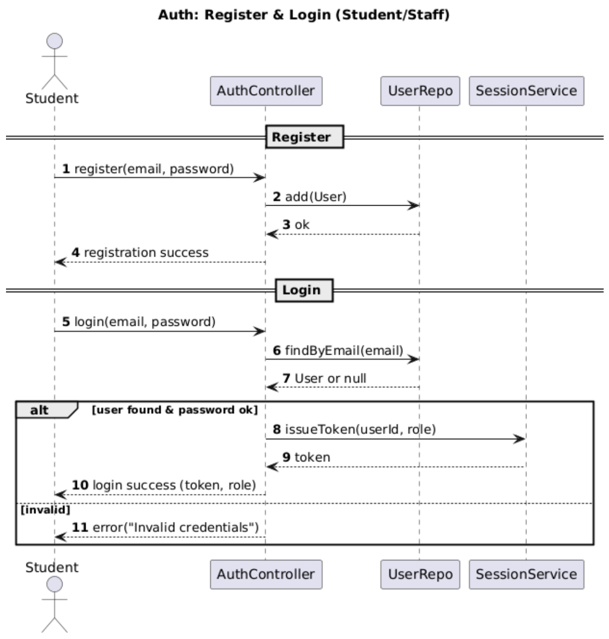
Design Patterns:

- `«strategy»` — *DueDateStrategy* (FixedDaysStrategy, CategoryBasedStrategy)
- `«factory method»` — *StorageFactory* for repository creation
- `«controller»` — *ApprovalController* (central coordinator)
- `«pure fabrication»` — Repositories for persistence abstraction

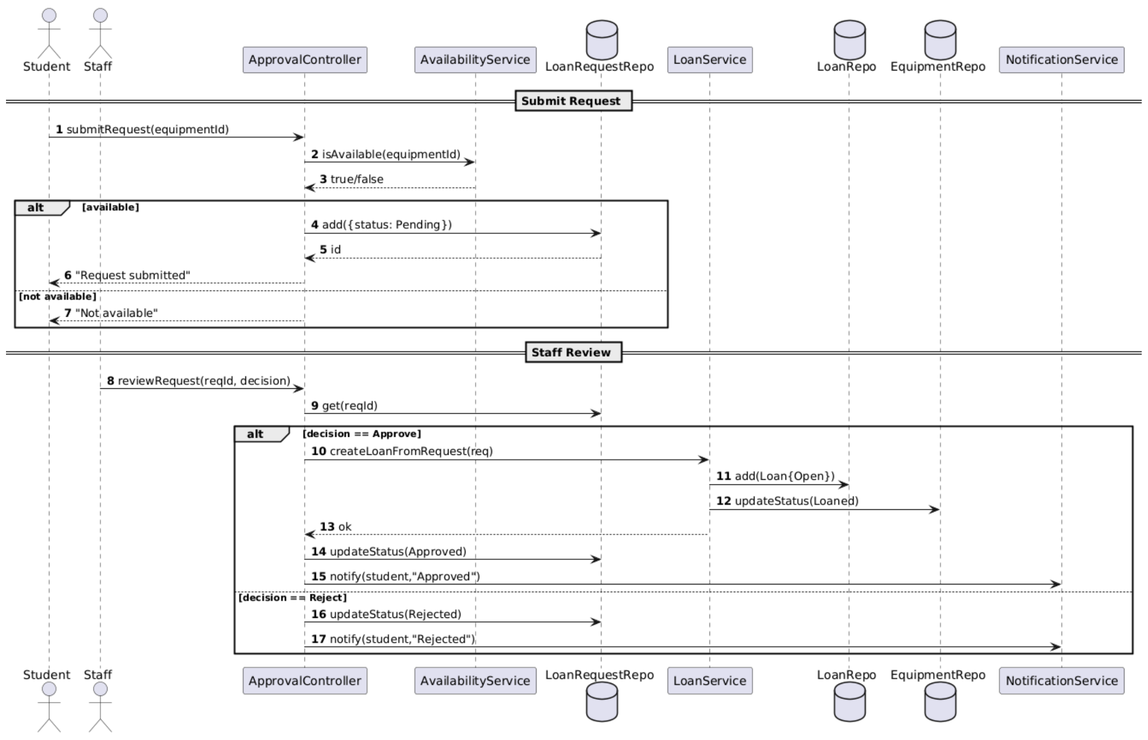


3.2 Sequence Diagram (Loan Request Flow)

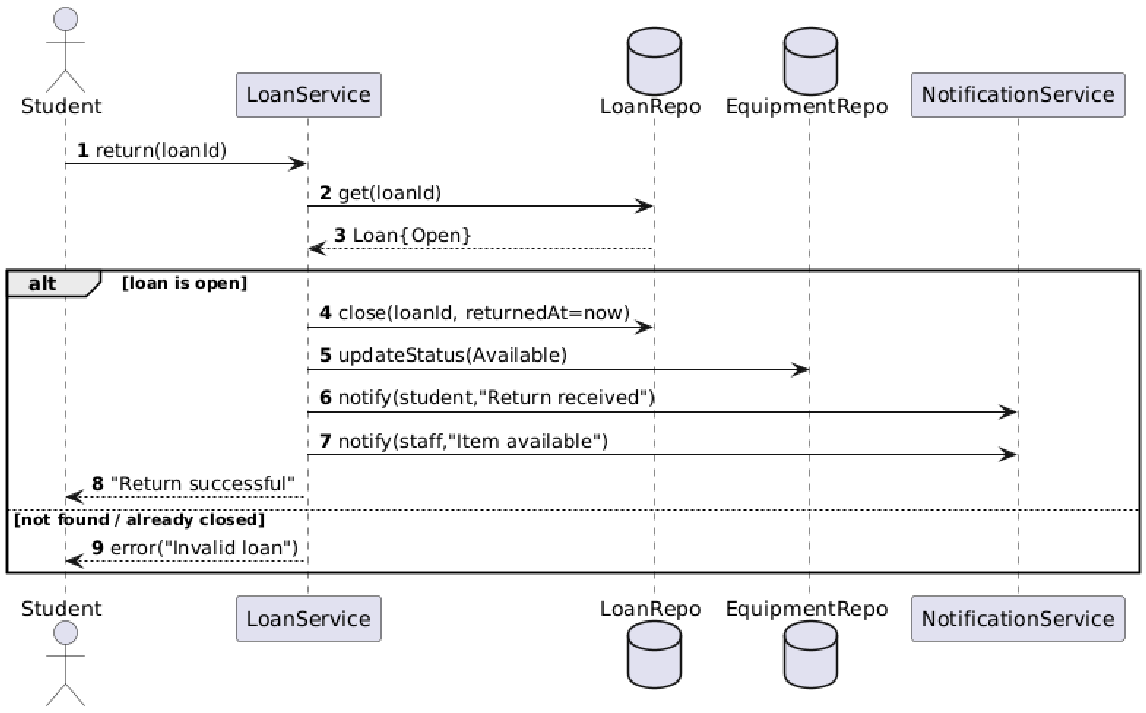
- Student → `ApprovalController.submitRequest()`
- Controller → `AvailabilityService.isAvailable()`
- Controller → `LoanRequestRepo.add(Pending)`
- Staff → `ApprovalController.reviewRequest(Approve/Reject)`
- [Approve] Controller → `LoanService.createLoanFromRequest()`
- `LoanService` → `LoanRepo.add()` → `EquipmentRepo.updateStatus(Loaned)`
- Notification → Student (status update)



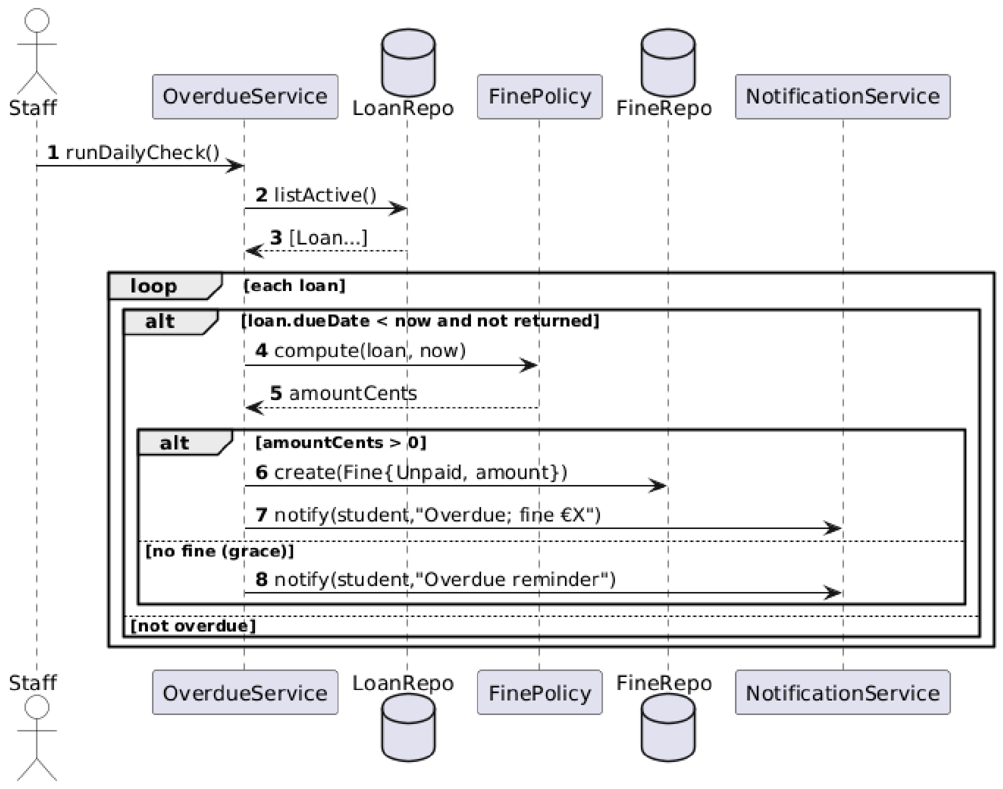
Loan Request & Approval (Student → Staff)



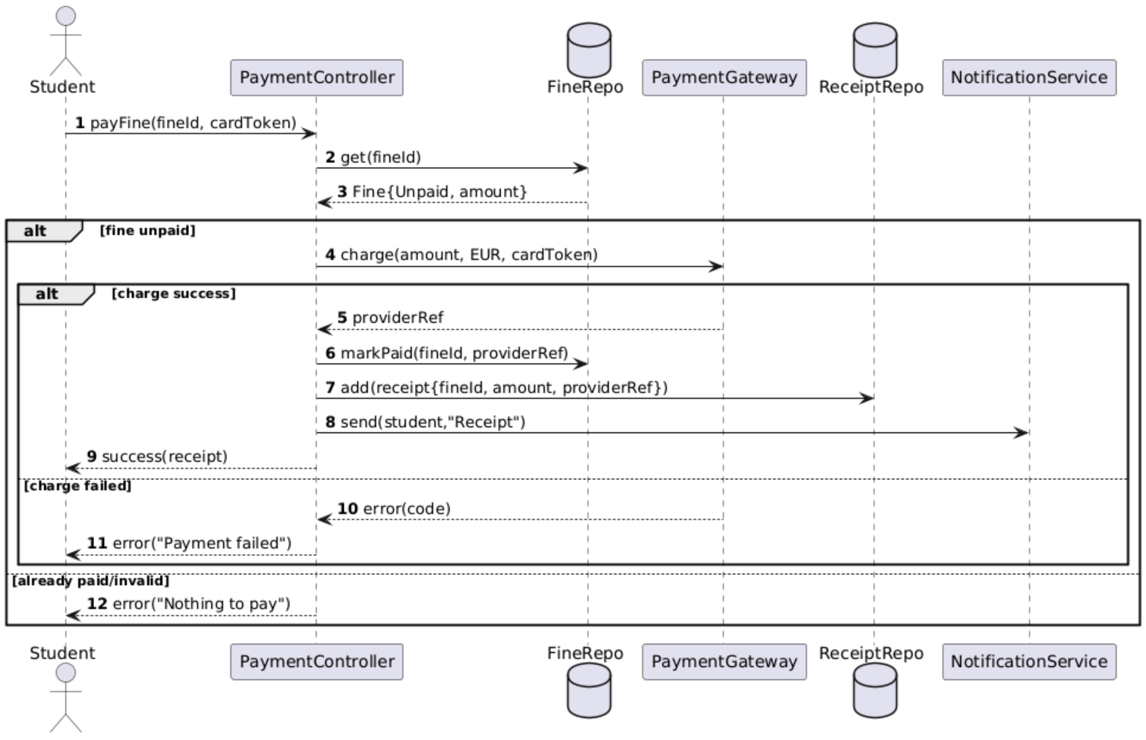
Return & Close Loan



Overdue Check + Fine Calculation (Scheduled job)



Online Payment of Fine



4. Anticipated Design Risks & Patterns

Risk / Challenge	Pattern Applied	Benefit
Changing storage (memory → DB → API)	Factory Method + Repository (DIP)	Decouples storage from business logic; enables easy migration
Varying due-date rules per category or semester	Strategy Pattern	Add new policies without editing existing code (OCP).
Controllers doing too much UI logic	GRASP Controller + Pure Fabrication (Services)	Clear responsibilities; maintainable code and tests.
Online payment gateway integration	Adapter Pattern	Swap between mock and live gateways safely

Implementation Notes:

- Strategy: DueDateStrategy (FixedDaysStrategy, CategoryBasedStrategy)
- Factory Method: StorageFactory.create(...) for repository instances
- GRASP Controller: ApprovalController centralizes main flows
- Pure Fabrication: Repository layer isolates persistence logic

5. Design Principles Applied

- **Single Responsibility:** Controllers coordinate; services apply logic; repositories persist data
- **Open/Closed:** New policies or gateways plug in via Strategy/Factory without code modification
- **Dependency Inversion:** High-level modules depend on abstractions, not concrete implementations

GRASP:

- *Controller:* `ApprovalController`
- *Information Expert:* Domain entities (Loan, Equipment)
- *Low Coupling / High Cohesion:* Repository and service design

6. Design Patterns Applied

Strategy — `DueDateStrategy`
Reason: Different categories may have different return policies; Strategy enables flexible extension.

Factory Method — `StorageFactory`
Reason: Allows switching storage (in-memory, DB, or API) without breaking core logic.

Adapter — `PaymentGatewayAdapter`
Reason: Handles integration with payment APIs or mock gateways uniformly.

GRASP Controller — `ApprovalController`
Reason: Centralizes request/approval logic and separates UI from domain logic.

Pure Fabrication — Repository Layer
Reason: Keeps persistence logic separate from business rules for cleaner code and easier testing.*

Summary:

This updated version keeps the project **clean and consistent with only two roles (Student and Staff)**.

It maintains full feature coverage (loans, fines, payments, reports) while simplifying user management.

The unified Staff role still respects SOLID and GRASP principles by isolating each responsibility through service classes and design patterns.