Lubelska Akademia WSEI
w Lublinie

Course: *Advanced Software Engineering°*

Laboratory 01

Design Patterns for University Access Control Application

Author:	Date:	Group:
Feranmi Akinwale	30-10-2025	Gr-1

Team Members:
Feranmi Akinwale

Supervisor:
dr inż. Tomasz Giżewski

Contents

1 Introduction **2**

2 Conceptual Functionalities **2**

2.1 Access Control . . . 2

2.2 Work Time Management . . . 2

2.3 Room Occupancy and Scheduling . . . 3

3 Applied Design Patterns **3**

3.1 Singleton Pattern - Central Configuration Manager . . . 3

3.1.1 Singleton — Class Diagram . . . 3

3.2 Factory Method Pattern - Dynamic User Creation . . . 4

3.2.1 Factory Method — Class Diagram . . . 4

3.3 Observer Pattern - Real-Time Notification System . . . 5

3.3.1 Observer — Class Diagram . . . 5

4 Sequence Diagram: Card Scan / Authorization Flow **6**

5 NS Diagrams **7**

5.1 NS 1 - AccessPolicyManager.getInstance() . . . 7

5.2 NS 2 - RoleUserFactory.createUser(role, id) . . . 7

5.3 NS 3 - Subject.notify(evt) . . . 7

1. Introduction

This laboratory project presents the conceptual design of a *University Access Control and Scheduling System*. The purpose is to manage user authentication, building access, work-hour logging, and classroom scheduling within campus environment. To ensure scalabiltiy and maintainability, three behavioral design patterns are used:

- **Singleton** - This is to guarantee a single global configuration manager that holds universal policies such as access levels and logging preferences.
- **Factory Method** - This is to dynamically create various user roles (Administrator, Lecturer, Student, Technician, Visitor) based on authentication data.
- **Observer** - This is to enable real-time notifications so dashboards and monitoring interfaces automatically react to events such as card scans or room occupancy changes.

  Together, these patterns form a coherent architecture that separates concerns, promotes extensibility, and supports reactive behavior across system modules.

2. Conceptual Functionalities

2.1. Access Control

Every person entering the campus is associated with a digital identity card. When a card is scanned at the entrance terminal, the system authenticates the user and consults the central **Access Policy Manager** (SIngleton). If access is granted, the event is logged by the global logger and observers such as security dashboard and audit modules are notified in real time.

2.2. Work Time Management

University employees (lecturers, administrators, technicians) registers their start and end times by scanning their cards. The **Work Time Service** communicates with a reporting module that aggregates daily working hours. The Observer pattern ensures that any change in recorded hours instantly updates connected interfaces, such as HR dashboards or employee portals.

2.3. Room Occupancy and Scheduling

Each lecture room has an occupancy sensor or schedule trigger. When a room becomes occupied, an event is broadcast to observed subscribers - classroom dashboards, timetables apps, or the central monitoring screen. Scheduling modules employ the Factory Method to create appropraite schedule objects (*LectureSchedule, ExamSchedule, MaintenanceSchedule*), enabling flexible handling of different event types.

3. Applied Design Patterns

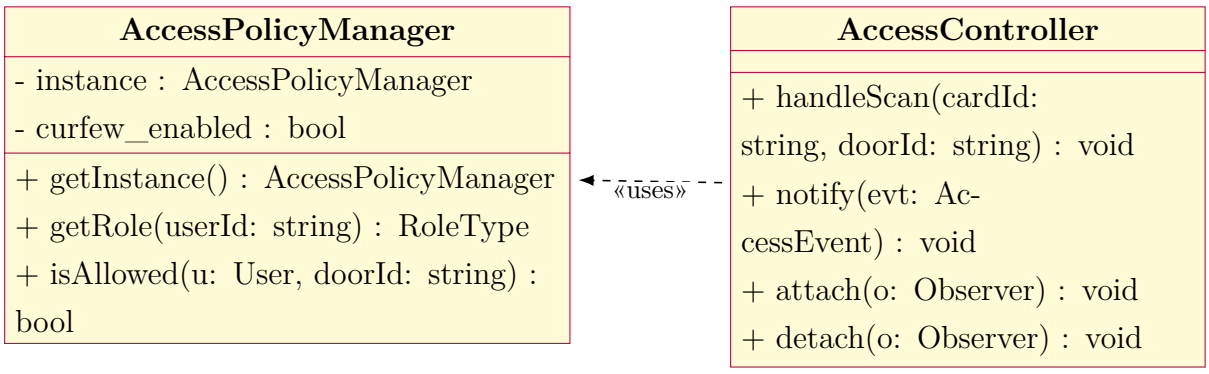
3.1. Singleton Pattern - Central Configuration Manager

The **Singleton pattern** is used to ensure that certain component of the system exists only once during runtime. In this architecture, it is applied to the **AccessPolicyManager**, which acts as a central authority for all authentication and authorization rules across the university system.

Whenever a user scans an identification card, any module that requires policy verification retrieves the same instance of the AccessPolicyManager through its `getinstance()` method. This guarantees that all parts of the system rely on a single, coherent set of rules rather than maintaining separate or conflicting copies.

The Singleton also improves maintainability and resource efficiency. Configuration data, logging preferences, and global parameters are stored and updated in one location, ensuring consistency throughout the platform. Because only one instance exists, it becomes easier to synchronize access, maintain system state, and reduce memory overhead.

3.1.1. Singleton — Class Diagram



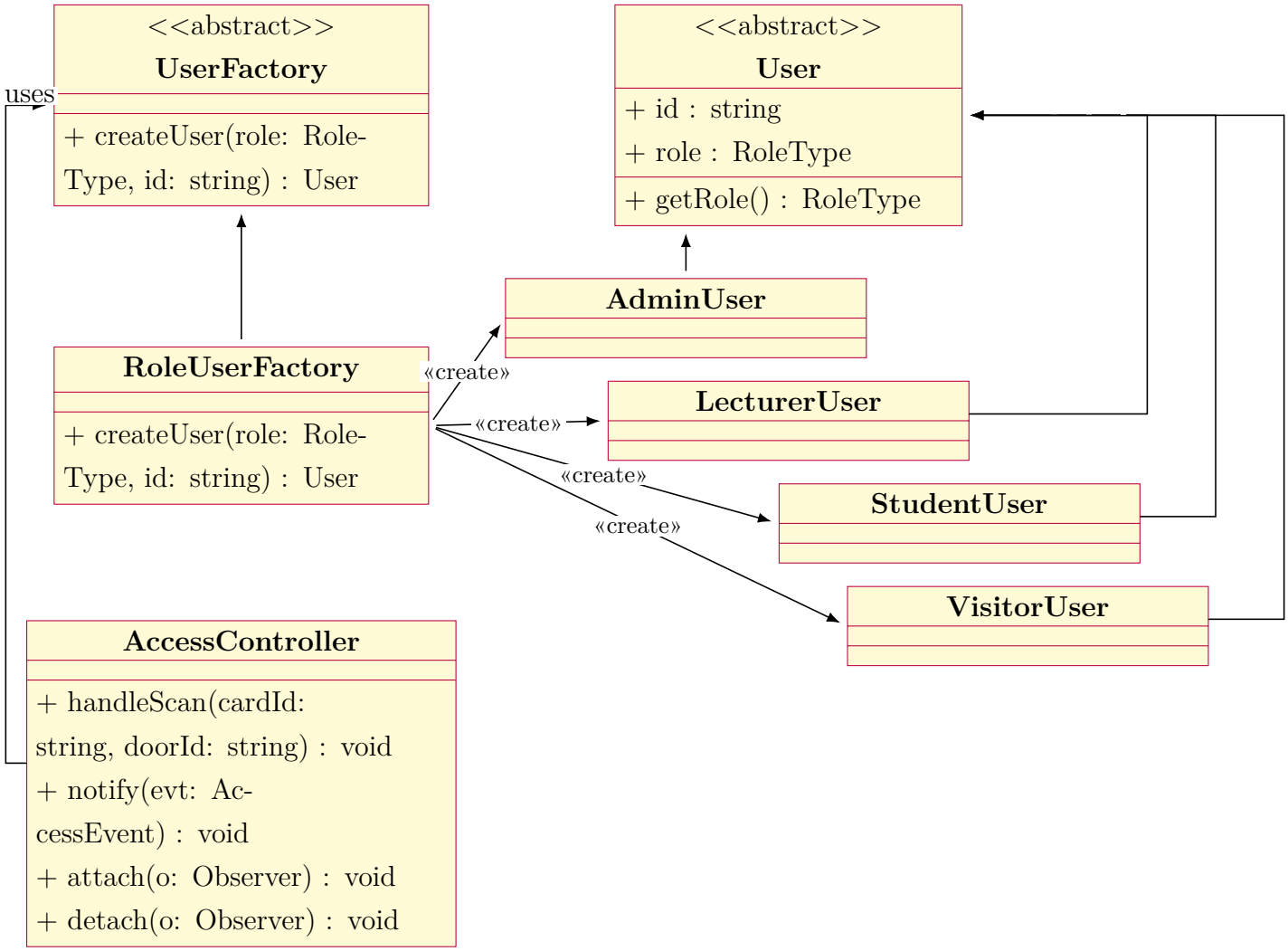
Rys. 1. Singleton pattern: one `AccessPolicyManager` used by `AccessController`.

3.2. Factory Method Pattern - Dynamic User Creation

The **Factory Method pattern** is employed to handle the creation of user objects with different roles and permissions. Instead of hard-coding conditions to instantiate classes like `AdminUser`, `LecturerUser`, or `StudentUser`, the system defines an abstract creator, `UserFactory`, that declares a method `createUser(roleType)`. A concrete factory (`RoleUserFactory`) decides which user class to instantiate.

Adding a new role such as `VisitorUser` requires only extending the factory, adhering to the Open–Closed Principle.

3.2.1. Factory Method — Class Diagram

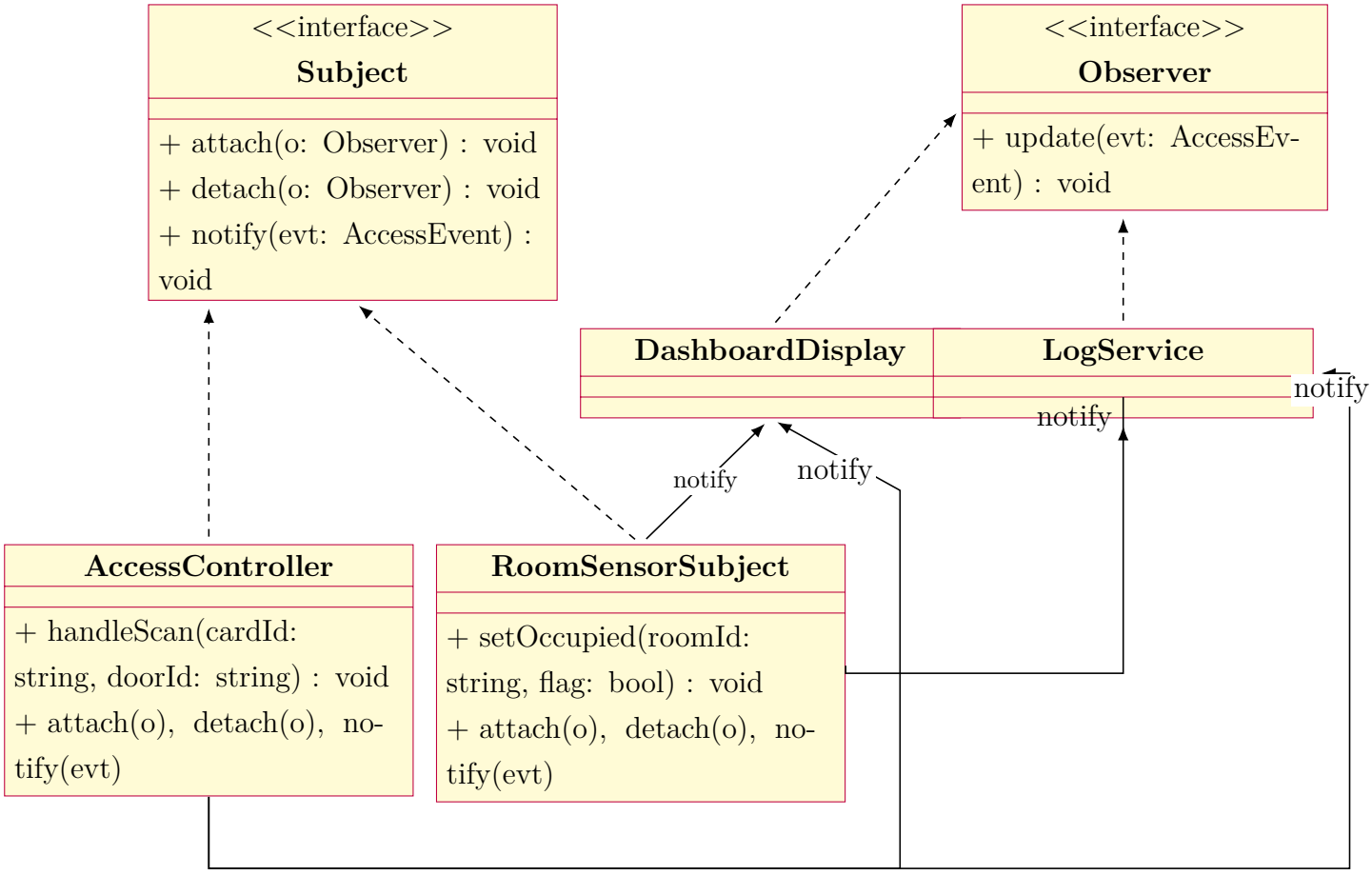


Rys. 2. Factory Method: `UserFactory` (creator) → `RoleUserFactory` (concrete creator) producing concrete `User` types.

3.3. Observer Pattern - Real-Time Notification System

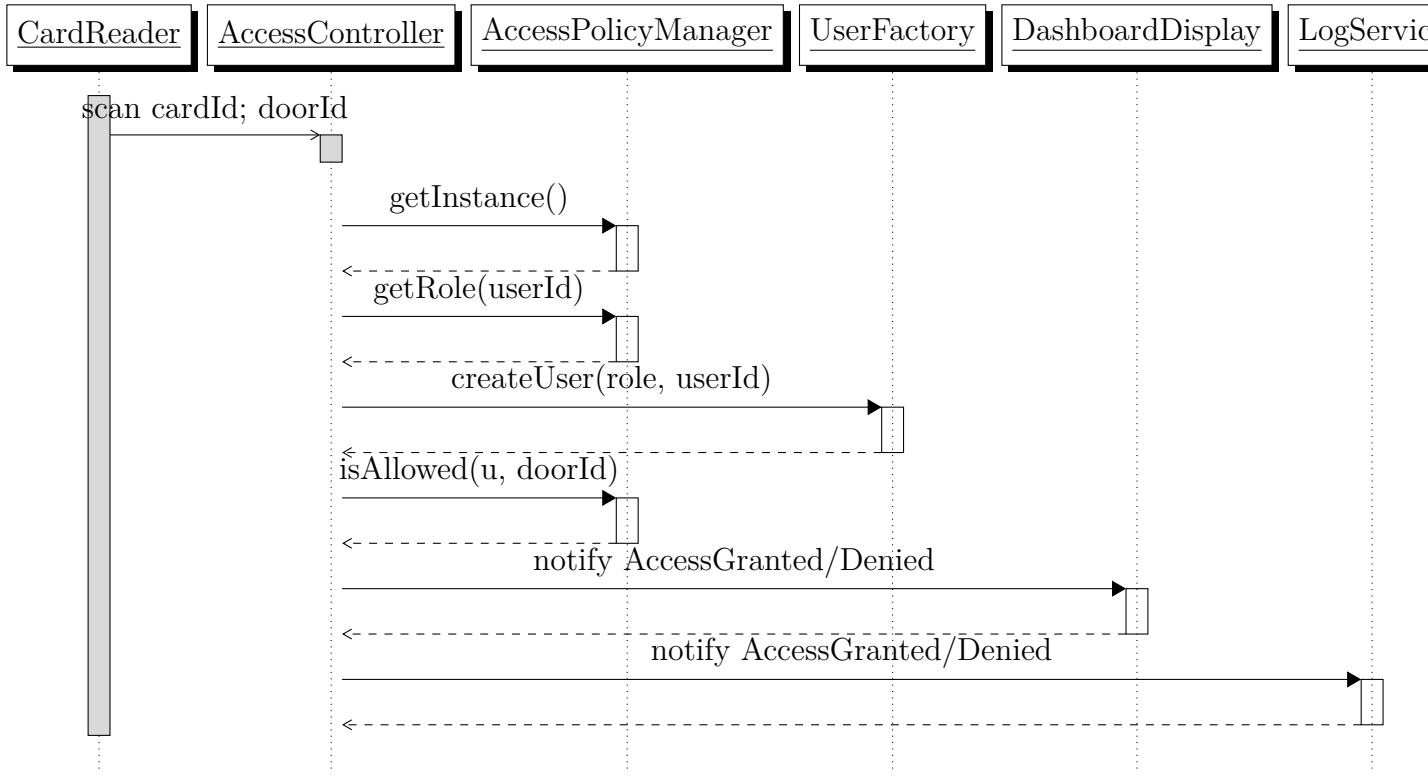
The **Observer pattern** manages the flow of real-time updates within the system. It defines a one-to-many relationship between a *Subject* (the event source) and *multiple Observers* (the components that respond to those events). In this project, **AccessController** and **RoomSensorSubject** act as subjects. Observers such as `DashboardDisplay` and `LogService` subscribe via `attach()/detach()`. When a subject calls `notify()`, all observers update automatically. This loose coupling allows new observers to be added without altering existing code.

3.3.1. Observer — Class Diagram



Rys. 3. Observer: Subjects (`AccessController`, `RoomSensorSubject`) implement `Subject` and notify Observers (`DashboardDisplay`, `LogService`) that implement `Observer`.

4. Sequence Diagram: Card Scan / Authorization Flow

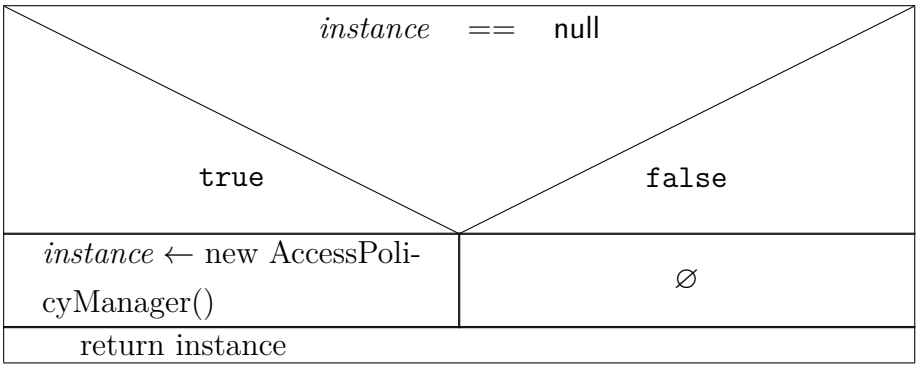


Rys. 4. Sequence Diagram: AccessController handles a scan, consults the Singleton policy, creates the User via Factory Method, and notifies Observers.

The sequence diagram illustrates the interaction flow for a card scan authorization process within the University Access Control System. When a user scans their ID card, the `AccessController` orchestrates the entire operation by retrieving the centralized policy instance from the `AccessPolicyManager` (Singleton pattern), identifying the user's role, and dynamically creating the appropriate user object through the `UserFactory` (Factory Method pattern). The controller then verifies access permissions and notifies subscribed observers such as the `DashboardDisplay` and `LogService` (Observer pattern) about whether access was granted or denied. This unified flow demonstrates how the three design patterns collaborate to maintain modularity, scalability, and real-time responsiveness in the system.

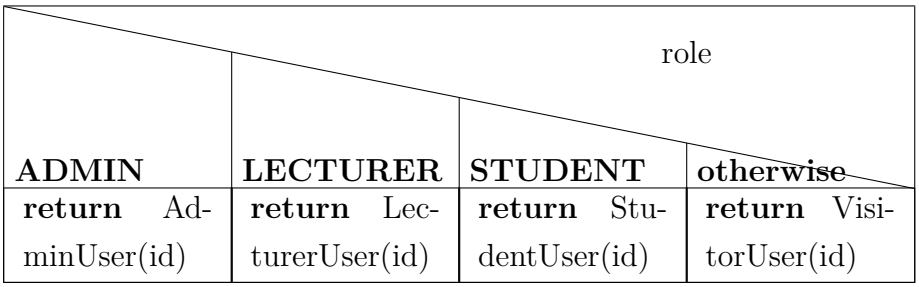
5. NS Diagrams

5.1. NS 1 - AccessPolicyManager.getInstance()



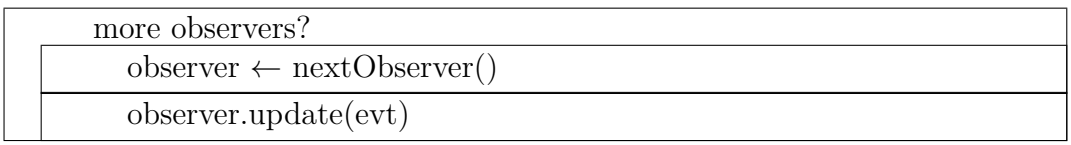
Rys. 5. NS diagram: Singleton `getInstance()`. If no instance exists, create it; then return the single instance.

5.2. NS 2 - RoleUserFactory.createUser(role, id)



Rys. 6. NS diagram: Factory Method `createUser(role, id)` returning the concrete `User`.

5.3. NS 3 - Subject.notify(evt)



Rys. 7. NS diagram: Observer `notify(evt)` iterates through subscribers and calls `update`.

Bonus: Prototype Demonstration

The complete Python implementation demonstrating the three behavioral patterns is available at: `https://github.com/Oluwaferanmiii/patterns_demonstration`