Lubelska Akademia WSEI
w Lublinie

Course: *Advanced Methods of Application Engineering*

# Laboratory Instruction

## Design Patterns and their UML Representation

| Author: | Date: | Group: |
|---|---|---|
| *dr. eng. Tomasz Giżewski* | 1-10-2024 | ∅ |

Team Members:
WspółAutor Raportu

Supervisor:
dr. eng. Tomasz Giżewski

## 1. Introduction

Design patterns are reusable solutions to common problems encountered in software design. They provide a proven template for structuring code, making it easier to develop flexible, maintainable, and efficient systems. By abstracting best practices, design patterns help developers avoid reinventing the wheel and streamline the development process.

Each design pattern addresses a specific design challenge, such as object creation, communication, or structural relationships. Patterns are generally categorized into three groups: creational, structural, and behavioral. Creational patterns, such as Singleton and Factory Method, focus on object instantiation. Structural patterns, like Adapter and Composite, deal with organizing classes and objects into larger structures. Behavioral patterns, such as Observer and State, define how objects interact and communicate.

Using design patterns improves code readability and consistency, especially in large projects. They also facilitate collaboration by providing a common vocabulary for describing solutions. For example, saying "we'll use the Singleton pattern here"immediately conveys the intended design to experienced team members.

Although design patterns are powerful tools, they are not a one-size-fits-all solution. Overusing them or applying them inappropriately can lead to unnecessary complexity. Therefore, it is crucial to understand the problem thoroughly before selecting a pattern.

In modern software engineering, design patterns remain an essential skill for developers, bridging the gap between theoretical knowledge and practical implementation. By mastering them, developers can write cleaner, more efficient, and maintainable code, ultimately enhancing the quality of their software systems.

## 1.1. Singleton

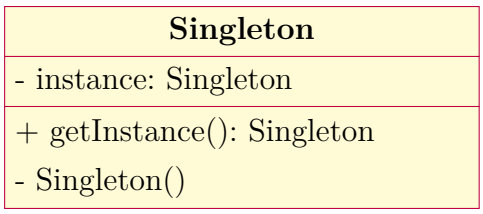
The **Singleton** pattern is one of the fundamental creational design patterns. Its main purpose is to ensure that a class has **only one instance** throughout the entire application and that there is a **global access point** to this instance. In practice, this means that no matter where in the code a developer refers to this class, they will always get *the same object*. This approach is commonly used when:

- the class manages a **shared resource**, such as a database connection, configuration file, logger, or settings manager,
- creating multiple instances would be inefficient (e.g., due to costly initialization),
- a consistent application state must be maintained across different components.

The structure of a Singleton class is based on three key principles:

1. **Private constructor** — prevents creating new objects from outside the class using the `new` operator. This guarantees that the only way to obtain an instance is through a dedicated static method.
2. **Private static instance field** — stores the single instance of the class. It is initialized the first time the accessor method is called.
3. **Static `getInstance()` method** — returns a reference to the existing instance, and if none exists yet, it creates one and stores it in the static field.

In object-oriented notation, the structure of the Singleton pattern can be represented in UML as follows:




Rys. 1. UML class for singleton pattern

The operational logic of the `Singleton` pattern is remarkably simple yet conceptually elegant. When the static method `getInstance()` is invoked for the first time, the program begins by checking whether the internal static field named `instance` already holds a reference to an existing object. If this field is still `null`, meaning that no instance has yet been created, the method proceeds to construct a new object of the class `Singleton`. This freshly created instance is then stored in the static field, effectively registering itself as the single representative of that class within the application's runtime environment. From that moment onward, every subsequent

invocation of `getInstance()` simply returns this same, already established instance. Thus, the method becomes both a factory and a gatekeeper — it creates the object once, and thereafter guards exclusive access to it, ensuring that the class remains truly unique across the entire program execution.

```
class Singleton {
  private static Singleton instance;
  private Singleton() {
  // private constructor-cannot be accessed from outside
  }

  public static Singleton getInstance() {
    if (instance == null) {
      instance = new Singleton();
    }
    return instance;
  }
}
```

Listing 1 Example in pseudocode

In modern JavaScript, the Singleton pattern finds its natural place within the module system introduced in ECMAScript 2015. Unlike in older versions of the language, where global variables and function closures were often misused to simulate single instances, the ES6 module loader ensures that each module is evaluated only once and cached for subsequent imports. This elegant mechanism means that the Singleton pattern is, in a sense, built into the language itself.

A typical example of such a modern implementation can be seen in a simple `Logger` class. The class defines a constructor which first checks whether an instance of itself already exists. If it does, the constructor returns that same instance, thus ensuring that the object cannot be duplicated. Inside, the logger maintains an internal array of log entries and a method that prints them to the console with timestamps.

```
class Logger {
  constructor() {
    if (Logger._instance) {
      return Logger._instance;
    }

    this.logs = [];
    Logger._instance = this;
    }

  log(message) {
    const timestamp = new Date().toISOString();
    this.logs.push({ message, timestamp });
    console.log(`[${timestamp}] ${message}`);
  }
```

```
16
17   getLogCount() {
18     return this.logs.length;
19   }
20 }
21
22 const logger = new Logger();
23 Object.freeze(logger);
24 export default logger;
```

Listing 2 ES6 Singleton

The code presented on (listing 2) where, the Singleton mechanism emerges in a particularly clear form. When the module is imported elsewhere in the application, for example as `import logger from './Logger.js';`, the JavaScript engine does not create a new `Logger` object. Instead, it provides the same frozen instance that was produced during the first evaluation of the module. Thus, all parts of the application share a single, coherent source of truth for logging events.

This pattern has several practical advantages. It removes the risk of duplicate loggers writing to different outputs and ensures that all system events are collected in one consistent sequence. By freezing the exported instance, the developer further enforces immutability — no external code can accidentally alter the logger's state or replace it with another implementation. Such discipline is especially valuable in complex applications built on frameworks like React, Angular, or Node.js services, where concurrency and asynchronicity might otherwise lead to subtle inconsistencies.

From a broader perspective, this approach reflects how JavaScript's evolution has integrated long-established design principles directly into its module architecture. The Singleton is no longer a special contrivance of language tricks but rather a straightforward idiom supported by the runtime itself. It is both concise and expressive — a small but elegant demonstration of how classical design patterns adapt naturally to modern programming paradigms.

Although the idea behind the Singleton pattern may appear deceptively straightforward, its practical implementation demands a certain degree of engineering discipline. The first and perhaps most critical consideration arises in the context of **multithreaded environments**. When multiple threads execute concurrently, there is a potential race condition: two threads might call `getInstance()` at nearly the same moment, both finding that the internal reference has not yet been initialized. Without appropriate synchronization, this can lead to the creation of two distinct objects — a clear violation of the Singleton's foundational principle. Therefore, techniques such as synchronized blocks in Java, `lock` statements in C#, or the use of thread-safe constructs like *double-checked locking* are employed to guarantee that only one thread performs the initialization.

Another subtle but elegant refinement is known as **lazy initialization**. Instead of constructing the Singleton eagerly at program startup — when it might not even be needed — the object is created only upon its first request. This approach conserves resources, especially when initialization involves complex operations such as opening database connections or reading configuration files. In many modern languages, this behavior can be expressed concisely using a `static initializer` or language-level support for lazy evaluation, ensuring both efficiency and thread safety without additional synchronization overhead.

Finally, a word of caution is in order. The Singleton pattern, though conceptually neat, is among the most frequently *overused* design constructs. Because it provides easy global access, developers sometimes treat it as a convenient shortcut to share data or services across components. However, such convenience often comes at the cost of architectural clarity. Excessive reliance on Singletons can lead to hidden dependencies, reduced modularity, and difficulties in unit testing — particularly because global state is notoriously hard to isolate or mock in controlled testing environments. Thus, the pattern should be employed deliberately and sparingly, reserved only for those cases where global uniqueness is truly essential.

In well-structured software systems, legitimate applications of the Singleton pattern can be found in several recurring scenarios. A classic example is the **global event logger**, which records messages, warnings, and diagnostic information across an entire application. Another common use is the **database connection manager**, where a single instance maintains shared access to a resource that must remain unique and synchronized. Singletons also frequently serve as **configuration handlers**, encapsulating parameters read from files such as `config.json` and exposing them uniformly throughout the system. In lower-level or embedded contexts, a Singleton may represent a **central communication interface** with the operating system, a hardware controller, or any other unique subsystem that cannot be duplicated.

In all these cases, the Singleton acts not merely as a technical convenience, but as a conceptual guarantee: it enforces the existence of exactly one, consistent, and authoritative instance, thereby providing structural coherence to the software architecture as a whole.

## 1.2. Factory Method

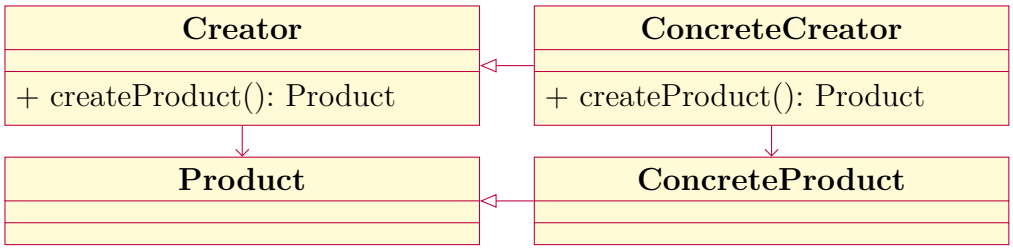
The **Factory Method** pattern represents one of the most fundamental and intellectually satisfying ideas in object-oriented design — the separation of *object creation* from *object usage*. Its guiding principle is deceptively simple: define a common interface for creating objects, but allow subclasses to decide which concrete class should actually be instantiated. In doing so, the pattern transfers the responsibility of choosing the product's type from the base class to its specialized descendants.

At first glance, this may appear as a mere structural convenience. However, it carries profound architectural implications. By delegating object creation to subclasses, we achieve a

high level of flexibility and extensibility: the code that relies on products (that is, objects being created) remains blissfully unaware of their concrete implementation. The client code merely calls the factory method, trusting that it will receive a valid object conforming to a known interface. This decouples the *what* from the *how* — the abstract intention from the concrete realization.

The essence of the pattern is captured in a minimal hierarchy. An abstract base class, often called the `Creator`, declares the factory method, for instance `createProduct()`. This method specifies the contract for object creation but leaves the actual instantiation to the subclasses. Each subclass, such as `ConcreteCreator`, overrides the factory method to produce a specific type of product — say, an instance of `ConcreteProduct`. Both the abstract `Product` and its specialized version form a parallel hierarchy on the receiving end of this relationship.

The UML diagram below illustrates this structure in its canonical form.




Rys. 2. UML diagram for Factory Method Pattern

To understand how this works in practice, let us imagine a system designed to produce different types of documents — for instance, text reports, spreadsheets, or PDF exports. Rather than scattering conditional statements throughout the code (`if`–`else` chains deciding which class to instantiate), we define an abstract `DocumentCreator` class that declares a method `createDocument()`. Each subclass, such as `PDFCreator` or `ExcelCreator`, implements this method to return the appropriate document type. The client interacts only with the abstract creator interface, and yet the resulting document type dynamically depends on which concrete subclass has been invoked. In effect, the pattern encapsulates the entire object-creation process behind a clean, extensible boundary.

This separation of concerns brings several important advantages. First, it allows new product types to be introduced without modifying existing client code — the very essence of the *Open–Closed Principle*. Second, it promotes consistency across families of related classes, ensuring that each creator produces only objects belonging to its designated product hierarchy. Finally, it makes testing and maintenance easier, as dependencies are reduced to interfaces rather than concrete implementations.

In modern frameworks and architectures, the Factory Method underlies many familiar mechanisms: GUI toolkits use it to create platform-specific controls; database libraries rely on it

to instantiate appropriate drivers; and dependency injection containers effectively generalize it, transforming object creation into a configurable service. Thus, while the pattern itself is simple, its conceptual reach extends deep into the foundations of contemporary software engineering.

In contemporary JavaScript, the Factory Method pattern fits naturally into the class-based model introduced with ECMAScript 2015. By combining inheritance and polymorphism, we can define a flexible hierarchy where the act of object creation is delegated to subclasses. This design brings order to systems that must produce families of related objects — for instance, different types of user interfaces, file exporters, or data parsers — without overloading the code with conditional logic.

Let us consider a simple yet illustrative example. Imagine that our application needs to generate different types of *documents* — text-based reports and PDF summaries — depending on the selected output format. Instead of hardcoding these decisions throughout the program, we define an abstract creator class `DocumentCreator` with a method `createDocument()`. Each subclass then decides which specific document type it will produce.

```
// Abstract creator
class DocumentCreator {
  createDocument() {
    throw new Error("Abstract method must be overridden.");
  }

  printDocument() {
    // Common operation for all creators
    const doc = this.createDocument();
    console.log(`Printing document: ${doc.getType()}`);
  }
}

// Concrete product interface
class Document {
  getType() {
    throw new Error("Abstract method must be overridden.");
  }
}

// Concrete products
class PDFDocument extends Document {
  getType() {
    return "PDF Document";
  }
}

class TextDocument extends Document {
  getType() {
    return "Text Document";
```

```
  }
}

// Concrete creators
class PDFCreator extends DocumentCreator {
  createDocument() {
    return new PDFDocument();
  }
}

class TextCreator extends DocumentCreator {
  createDocument() {
    return new TextDocument();
  }
}

// Client code
const creators = [new PDFCreator(), new TextCreator()];
creators.forEach(c => c.printDocument());
```

The elegance of this design lies in its dynamic delegation of responsibility. The abstract class `DocumentCreator` provides the common behavior — in this case, the operation `printDocument()` — while leaving the actual instantiation of the product to its subclasses. Each subclass, such as `PDFCreator` or `TextCreator`, implements the `createDocument()` method, deciding which concrete product (a `PDFDocument` or a `TextDocument`) should be returned. The client code, on the other hand, operates solely on the abstract type `DocumentCreator`, without knowing or caring which particular product it is dealing with. Thus, the logic of object creation is elegantly encapsulated, allowing the system to be extended with new document types without modifying existing code.

This design has deeper implications beyond mere code structure. By centralizing the creation process in dedicated subclasses, we enforce consistency and remove conditional chaos from the program's core logic. It also aligns perfectly with the **Open–Closed Principle**: the system is open to extension (we can add new document types or creators) but closed to modification (existing classes remain untouched). Such an approach is emblematic of modern, maintainable architecture — one that privileges adaptability over rigidity.

In real-world ES6 applications, the Factory Method pattern often appears in frameworks and libraries: web components may use it to create platform-specific widgets; Node.js modules rely on it for configurable adapters; and front-end frameworks employ it to generate view components dynamically based on runtime data. Its enduring relevance stems from the fact that software, much like language itself, is constantly evolving — and a well-designed factory knows how to evolve gracefully.

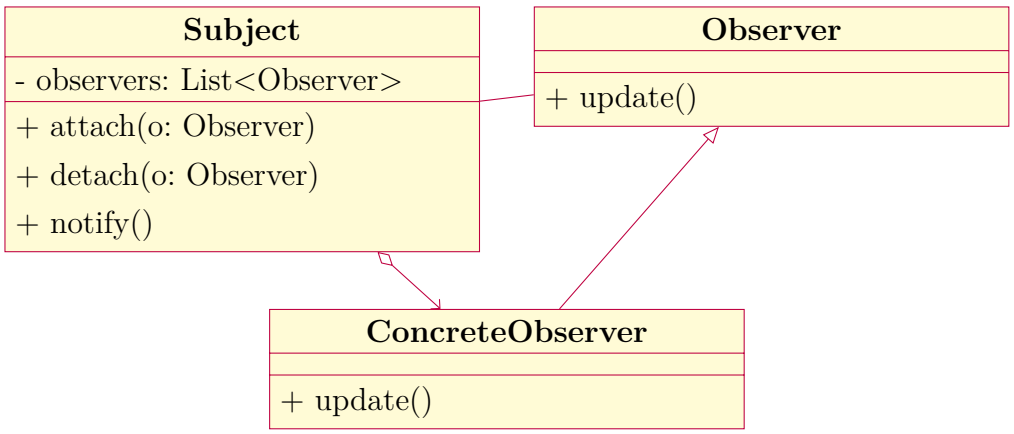
In summary, the Factory Method pattern teaches a subtle but crucial lesson: that the act of

creating an object should itself be an abstraction — one that evolves alongside the system rather than constraining it. It is a graceful answer to the tension between stability and flexibility — a hallmark of thoughtful software design.

## Observer

The **Observer** pattern belongs to the family of behavioral design patterns and expresses a subtle but powerful idea — that of establishing a *one-to-many relationship* between objects in a system. It provides a disciplined way for objects, known as *observers*, to automatically stay informed about changes in another object, known as the *subject*. Whenever the subject's internal state changes, all registered observers are notified and may update themselves accordingly. In essence, this pattern captures the concept of *reactivity* long before reactive programming became a buzzword.

The central figure in this collaboration is the `Subject`, which maintains a list of its dependent observers. It provides three essential operations: methods to `attach()` and `detach()` observers (thereby registering or unregistering them), and a `notify()` method that broadcasts updates to all currently registered observers. Each observer, in turn, must implement a standard interface — typically an `update()` method — which defines what should happen when it receives notification of a change. The key point is that the subject does not need to know *who* the observers are or what they do with the information; it simply announces that "something has changed," and the observers take it from there.




Rys. 3. The UML representation of the Observer Pattern

In practical software systems, this pattern arises naturally in any context where data changes must trigger actions in multiple dependent components. Consider, for example, a simple weather monitoring system. A central `WeatherStation` object (the subject) continuously collects temperature and humidity readings. Several display modules — such as a mobile app widget, a web dashboard, and an alert service — act as observers. Each subscribes to the weather station, and whenever new data arrive, the station calls their `update()` methods, passing along the rele-

vant measurements. The displays then refresh their readings independently, without the station needing to know their internal details or even how many displays exist.

This decoupling between subjects and observers offers a number of profound benefits. It enhances modularity, since both sides can evolve independently: new observers can be added without altering the subject's code, and vice versa. It improves scalability, as any number of observers can listen to the same subject. And it aligns beautifully with modern event-driven and reactive paradigms, where streams of data flow continuously between producers and consumers. Indeed, the Observer pattern forms the conceptual foundation of many contemporary frameworks — from the event emitters in Node.js, through GUI frameworks like JavaFX and Qt, to the reactive architectures of `RxJS` and modern front-end libraries such as React or Vue.

Yet, as with all elegant designs, this pattern must be applied thoughtfully. Without proper control, a large number of observers may introduce unwanted complexity or performance overhead, especially if each observer performs heavy computations upon receiving updates. Furthermore, cyclic dependencies — when observers in turn modify the subject — can lead to subtle and difficult-to-diagnose feedback loops. To mitigate such issues, engineers often employ buffering, event queues, or prioritization mechanisms to manage notification flow safely.

In summary, the **Observer pattern** embodies the principle of separation between *state* and *reaction*. It captures a deeply intuitive model of the world: that changes do not occur in isolation, but ripple outward, touching many interconnected parts of a system. Through this simple but expressive abstraction, software becomes not only functional, but responsive — a network of cooperating entities that, like participants in a well-orchestrated conversation, always remain in sync.

### Example in JavaScript (ES6)

In modern JavaScript, the **Observer** pattern naturally aligns with the event-driven character of the language. JavaScript applications — whether in the browser or on the server — are, by nature, reactive systems: they wait for events to occur and then respond to them. This makes the pattern not only familiar, but almost intrinsic to how JavaScript code is written today. By defining a clear relationship between subjects and their observers, we can bring order and structure to event propagation and state synchronization.

Let us consider a simple illustrative example — a minimalist event system that mimics the behavior of a weather station and its subscribers. We define a class `Subject` that maintains a collection of observers and provides three core methods: `subscribe()` to register new observers, `unsubscribe()` to remove them, and `notify()` to broadcast updates. Each observer must implement an `update()` method to react to the received information.

```
class Subject {
  constructor() {
    this.observers = new Set();
  }

  subscribe(observer) {
    this.observers.add(observer);
  }

  unsubscribe(observer) {
    this.observers.delete(observer);
  }

  notify(data) {
    this.observers.forEach(observer => observer.update(data));
  }
}

// Concrete observer
class WeatherDisplay {
  constructor(name) {
    this.name = name;
  }

  update(temperature) {
    console.log(`${this.name} reports temperature: ${temperature}
       in C`);
  }
}

// Client code
const station = new Subject();

const mobileApp = new WeatherDisplay("Mobile App");
const dashboard = new WeatherDisplay("Web Dashboard");

station.subscribe(mobileApp);
station.subscribe(dashboard);

station.notify(22.5);
station.notify(23.1);
```

When this program runs, both the `Mobile App` and the `Web Dashboard` receive temperature updates from the shared `Subject`. Each registered observer reacts independently, executing its own logic upon receiving the notification. From the subject's perspective, there is no knowledge of what the observers actually do — only that they exist and wish to be informed when the data change. This separation of concerns, where the subject is responsible solely for broadcasting events and

the observers handle their own reactions, represents the essence of the Observer pattern.

In this example, the choice of a JavaScript `Set` rather than a simple array is deliberate. It guarantees that each observer is unique and can be efficiently removed without worrying about duplicates. The code is therefore both semantically clear and computationally efficient — a small but meaningful refinement that reflects modern JavaScript's maturity.

From a broader perspective, this design exemplifies how the Observer pattern underpins modern event architectures. JavaScript frameworks such as React, Vue, and Angular all rely, in various forms, on this idea: components subscribe to data sources, and whenever the state changes, the view automatically re-renders. Even Node.js's built-in `EventEmitter` class embodies this same principle, allowing objects to emit and listen to events dynamically at runtime. Thus, the pattern is not a relic of academic design theory, but a living foundation of real-world programming practice.

Conceptually, the Observer transforms a program from a passive, sequential script into a *living system of reactions*. Each change becomes a message, each object a participant in an ongoing dialogue. It is through such subtle interactions that software acquires the quality we call responsiveness — a harmony between data, behavior, and user experience that defines modern interactive applications.

# 2. Project Assignment: University Access Control Application

The goal of this laboratory project is to design and conceptualize an architectural model for managing access, scheduling, and occupancy within a university campus environment. The exercise emphasizes the use of classical **behavioral design patterns** to structure communication, coordination, and control between system components. Each student will work individually, focusing on clarity of architecture, UML modeling, and correct application of patterns.

## 2.1. Project Objective

Students are to propose a conceptual design for a *University Access Control and Scheduling System* — a platform that supervises building access, class timetables, room occupancy, and working hours for various groups of users. The focus is not on coding, but on understanding how well-chosen design patterns can make a system more maintainable, extendable, and logically coherent.

## 2.2. Core Functionalities to Model

1. **Access Control**

- Model user authentication and permission management for academic, administrative, and technical staff, as well as students.
- Represent the interaction between the access subsystem and electronic identification cards.
- Define how access status changes propagate through the system (e.g., when a card is scanned).

2. **Work Time Management**
   - Model logging and verification of working hours for university employees.
   - Simulate communication between the monitoring service and the reporting module.
3. **Room Occupancy and Scheduling**
   - Show how multiple interfaces (e.g., dashboards, apps) react to changes in room occupancy.
   - Represent how schedules are updated and synchronized when classes are added, moved, or canceled.

# 3. Design Guidelines

At this stage, the goal is to build a conceptual and structural model using the following **behavioral design patterns**. Each student must choose and correctly apply all three of the following:

- **Singleton** — used to maintain global application configuration, e.g., access policy manager or central logging service. The student should identify one component that must have exactly one instance throughout the system.
- **Factory Method** — used for dynamic creation of different user roles or access permission objects. The factory should be able to generate, for example, `AdminUser`, `LecturerUser`, or `StudentUser` objects according to the provided criteria.
- **Observer** — used to implement real-time notifications when the system state changes. For instance, when a lecture room becomes occupied, all subscribed observers (dashboards, mobile panels, logs) should automatically receive an update.

Each design pattern must be justified in the report: students should explain *why* a particular pattern was chosen, *how* it fits into the logic of the system, and *what advantages* it brings in terms of modularity and scalability.

The conceptual design should be presented using:

- UML **class diagrams** illustrating relationships between components,
- UML **sequence diagrams** showing the behavior of the system (especially for Observer notifications),
- NS **structured diagrams** (using the `struktogram` environment in LaTeX) representing algorithmic logic of key methods such as `getInstance()` (Singleton), `createUser()` (Factory), and `notify()` (Observer).

## 4. Technical and Organizational Details

- The main focus is on architectural and conceptual modeling — no full implementation is required for passing the assignment.
- Students should use UML tools such as `draw.io`, `Lucidchart`, or any equivalent UML editor.
- Submissions that include an actual working implementation (in any modern programming language — e.g. Java, C#, JavaScript ES6, or Python) and a link to a hosted repository or live demo (e.g., GitHub, GitLab, CodeSandbox, or similar URL) will receive **bonus credit**.
- The implementation does not need to be complex — even a small demonstrative example showing how the three patterns interact in practice will be valued.

## 5. Submission Requirements

Each student must prepare and submit:

1. A structured project report in PDF format containing:
   - A description of the conceptual functionalities (access, scheduling, occupancy).
   - An explanation of the applied design patterns (**Singleton**, **Factory Method**, **Observer**) and their role in the architecture.
   - UML and NS diagrams clearly illustrating system structure and algorithmic logic.
2. A visual presentation (PDF slides) summarizing the model's design rationale.
3. **Bonus (premiated):** a URL linking to an implemented prototype demonstrating the three behavioral patterns in operation. The implementation may be in any language of choice; clarity and correctness of pattern usage are the key evaluation criteria.

## 6. Evaluation Criteria

Projects will be evaluated based on:

- Correct and meaningful use of the required design patterns.
- Logical consistency and architectural clarity of UML diagrams.
- Readability and precision of NS algorithmic diagrams.
- Quality of documentation and argumentation.
- (Bonus) Existence and clarity of the functional prototype available through a URL.

In summary, this assignment is designed to help students not only understand but also *apply* classical behavioral design patterns in a coherent, real-world context — building a conceptual model of a university access and scheduling system that mirrors the reactive, modular, and scalable nature of modern software architectures.