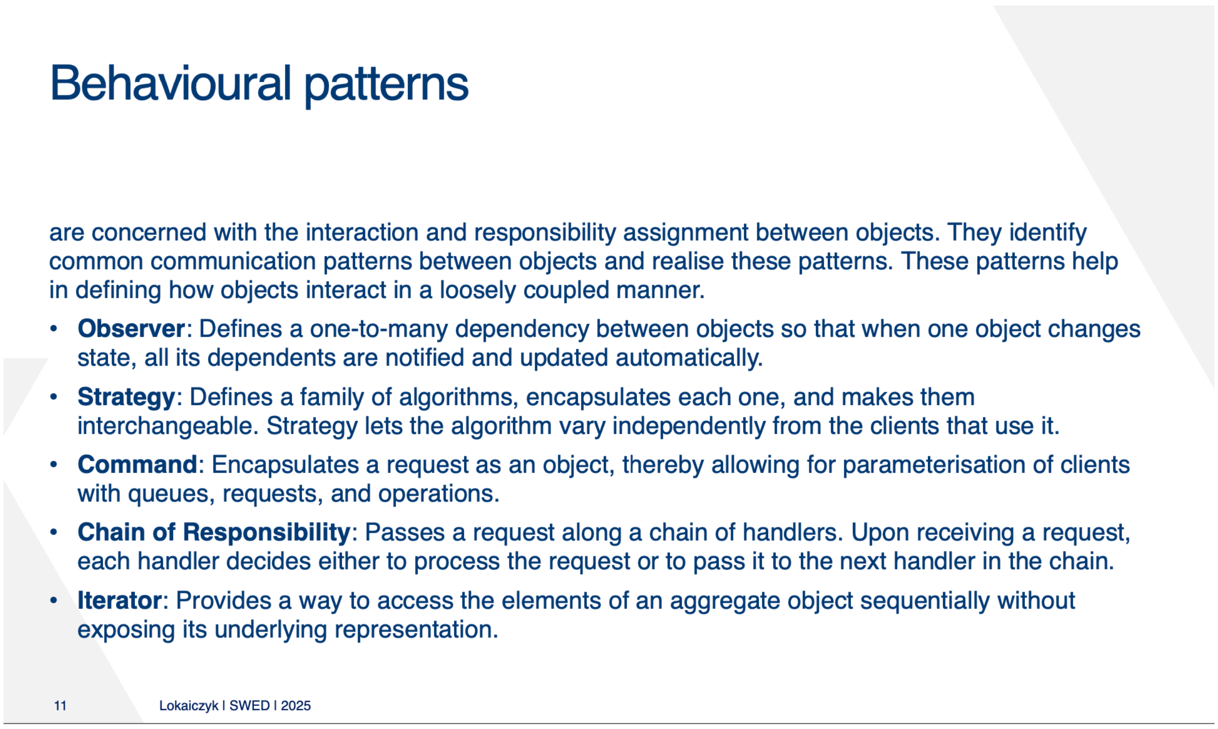
Exercise 5 - Website Monitoring System

Definition – Observer Pattern

The **Observer Pattern** is a design pattern in which a **subject** maintains a list of **observers** and notifies them automatically whenever its state changes.

This pattern enables **loose coupling** between the object that reports changes (the *Subject*) and the objects that react to those changes (the *Observers*).

Definition from the Lecturer:



Behavioural patterns

are concerned with the interaction and responsibility assignment between objects. They identify common communication patterns between objects and realise these patterns. These patterns help in defining how objects interact in a loosely coupled manner.

- **Observer**: Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **Strategy**: Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.
- **Command**: Encapsulates a request as an object, thereby allowing for parameterisation of clients with queues, requests, and operations.
- **Chain of Responsibility**: Passes a request along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.
- **Iterator**: Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

11 Lokaiczyk I SWED I 2025

Observer Pattern form in my Project:

- **Subject:** WebsiteMonitorService (monitors websites)
- **Observers:** User or Subscription objects that are informed when something changes

Previously, notifications were created and sent directly within the loop in the checkAllSubscriptions() method.

- **Clear separation of concerns:**
  - The **Observer** is only responsible for "**how** to react to updates."
  - The **Subject** is only responsible for "**how** to manage and notify its observers."
- This makes your code **much more flexible and easier to extend.**

Key Components:

- **Observer Interface** – defines how to react (e.g., update() method)
- **Subject Interface** – defines how to manage and notify observers

Task 2: My Class Diagram Explanation

This class diagram models a system where users can monitor websites and receive notifications when updates are detected. It uses the **Observer Pattern** for flexible and decoupled notification handling.

User Class

The User class represents a registered user oft he system. It contains attributes such as name, email, and phone number.

The User class also **implements the Observer interface** so it can receive notifications directly.

Subscription Class
Represents a user's subscription to a specific website.

Contains attributes like **subscriptionId** for unique identification, the **url** of the website, and an associated **NotificationPreference** instance for notification settings.
It does **not** directly store attributes like frequency or communication channel; instead, it delegates these settings to its **NotificationPreference**.

NotificationPreference Class

Defines how users want to be notified:

- Check frequency
- Communication channel (email, SMS, etc.)
- Message template

Each subscription has exactly one notification preference.

Notification Class

The **Notification** class is a simple container for messages. It holds all relevant information (– from which subscription it originates, when it was created, the actual message content, and through which channel it is delivered.)
The **WebsiteMonitorService** creates these notifications whenever a change is detected on a monitored website.

WebsiteMonitorService Class

Responsible for managing and storing user subscriptions using a Map<User, List<Subscription>>.

Monitors websites and generates notifications when changes are detected.

The method checkAllSubscriptions() iterates over all stored subscriptions to detect updates.

The WebsiteMonitorService **implements the Subject interface** to manage a list of observers (User objects).
Using attach(), detach(), and notifyObservers(), it updates all registered users when changes are found.

The system stores the user subscriptions and their notification preferences in a Map inside the WebsiteMonitorService class. Each user is associated with a list of subscriptions, and each subscription has exactly one notification preference.

Observer & Subject Interfaces

- **Observer**: Defines update(notification: Notification). Implemented by User.
- The Observer interface defines how objects receive updates from the subject when a change occurs, ensuring they react accordingly."
- **Subject**: Defines methods for managing observers (attach, detach) and notifying them (notifyObservers).