Exercise 6

First of all, what have I improved after presenting exercise 5 on Monday:

New Class WebsiteConentChecker:

I use a helper class WebsiteContentChecker to fetch the HTML content of a website. It uses a simple Scanner to read the content line by line and returns it as a string. This way, I can compare it to the previously stored version to detect real changes.

Improve WebsiteMonitorService Class:

In the WebsiteMonitorService, I store the previous content of each subscribed website in a Map. Every time the service checks for updates, it fetches the current HTML content using the WebsiteContentChecker, compares it with the stored version, and updates it if necessary. If a change is detected, a Notification is created.

Task 1. Rewrite your code from Exercise 5 so that it supports multiple strategies for website comparison.

1. Identical content size– gleiche Länge der Website (in Zeichen)

2. Identical html content– exakter HTML-Inhalt

3. Identical text content – nur der sichtbare Text, kein HTML-Code

I designed the system to support multiple comparison strategies. That way, I can choose how strictly or loosely website changes should be detected depending on the use case.

1. SizeComparisonStrategy

This strategy simply checks if the length of the website content has changed. If the new version has more or fewer characters than before, it's considered an update.

2.HtmlComparisonStrategy

This strategy compares the raw HTML of the website. Any change in the structure or elements of the HTML code will be detected.

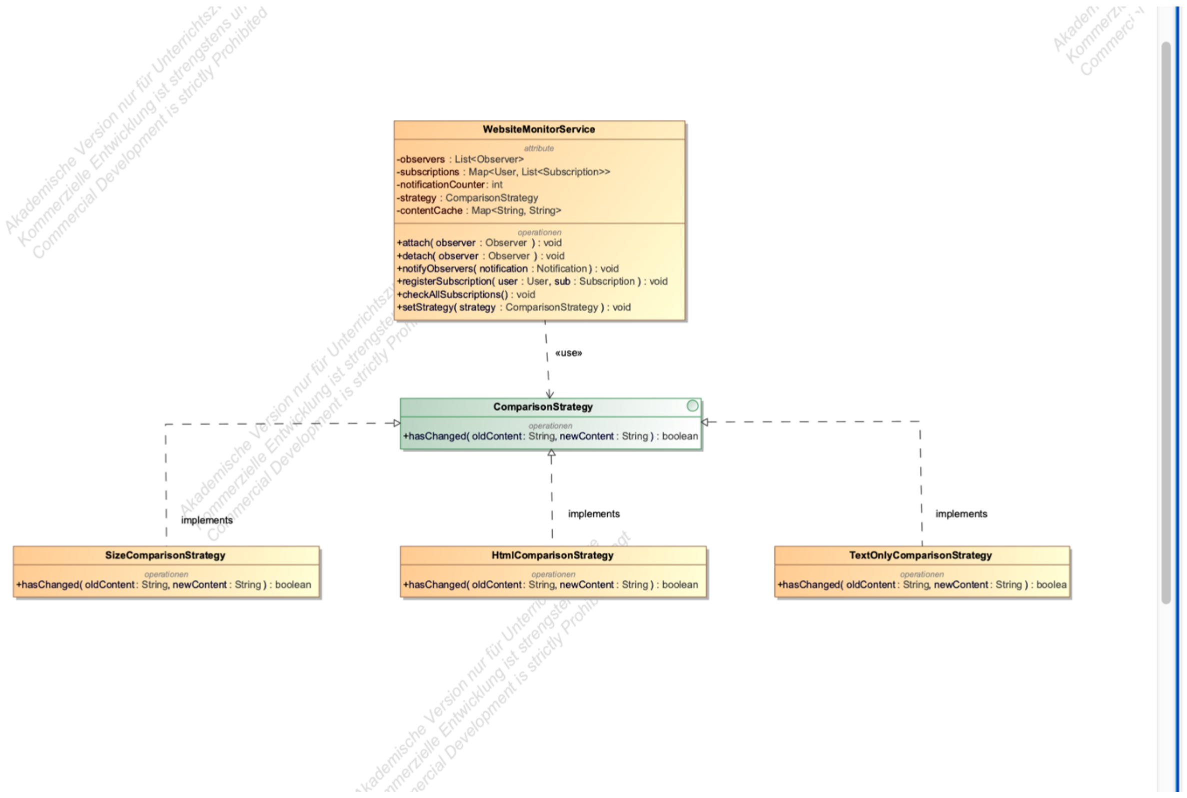
3. TextOnlyComparisonStrategy

In this strategy, all the HTML tags are removed first, so only the visible text remains. Then, it compares just the actual readable text — ignoring any changes in the underlying code.

Task 2: Please draw a UML diagram to illustrate the pattern.

This UML diagram shows how I used the **Strategy Pattern**.
The WebsiteMonitorService holds a reference to a ComparisonStrategy, which is an interface. I implemented three different strategies, each with its own logic to detect changes on a website. This makes my system flexible — I can switch strategies without changing the core logic

The other parts of the system like User, Observer, and Notification are not shown here to keep the diagram focused on the Strategy Pattern structure


WebsiteMonitorService
attribute
-observers : List<Observer>
-subscriptions : Map<User, List<Subscription>>
-notificationCounter : int
-strategy : ComparisonStrategy
-contentCache : Map<String, String>
operationen
+attach(observer : Observer) : void
+detach(observer : Observer) : void
+notifyObservers(notification : Notification) : void
+registerSubscription(user : User, sub : Subscription) : void
+checkAllSubscriptions() : void
+setStrategy(strategy : ComparisonStrategy) : void
«use»
ComparisonStrategy
operationen
+hasChanged(oldContent : String, newContent : String) : boolean
implements
implements
implements
SizeComparisonStrategy
operationen
+hasChanged(oldContent : String, newContent : String) : boolean
HtmlComparisonStrategy
operationen
+hasChanged(oldContent : String, newContent : String) : boolean
TextOnlyComparisonStrategy
operationen
+hasChanged(oldContent : String, newContent : String) : boolea

3. Describe the coding conventions you were using for your software.

In this project, I followed common Java coding conventions to ensure the clarity, maintainability, and readability of the code. These conventions guided the way classes, methods, variables, and code structures were written and organized.

Coding Conventions:

- **Naming Conventions**:
  Classes were named using PascalCase (e.g., User, WebsiteMonitorService), while variables and methods used camelCase
  (e.g., checkAllSubscriptions(), notificationCounter). This naming strategy makes it easy to tell different elements apart in the code.

- **Code Structure and Formatting**:
  The code follows consistent indentation (4 spaces), and brackets are always used for control structures, even when they enclose only one line. This improves readability and reduces the risk of logical errors.

- **Encapsulation**:
  All class attributes were kept private and accessed via public getters and setters. This ensures data integrity and encapsulation, which is a key principle of object-oriented design.

- **Single Responsibility Principle (SRP) Separation of Concerns**:
  Each class was designed with a single responsibility in mind. For example, WebsiteContentChecker is responsible only for retrieving website data, while WebsiteMonitorService handles monitoring and logic.

- **Use of Interfaces and Design Patterns** (**Interface Segregation Principle (ISP)**) :
  Interfaces like Observer, Subject, and ComparisonStrategy were used to implement well-known design patterns. This increases the flexibility and extensibility of the software by allowing different behaviors to be injected or changed easily.

- **Commenting and Documentation**:
  Comments were added in areas where the logic might not be immediately obvious, particularly in methods involving content fetching or comparison strategies.

- **Dependency Inversion Principle (DIP)**:
  High-level classes depend on abstractions, not concrete implementations: WebsiteMonitorService uses the ComparisonStrategy interface, so it is not tightly coupled to any specific comparison logic. This improves flexibility and makes unit testing easier by allowing mock strategies.

- **Open/Closed Principle (OCP)**:
  The system is open to extension but closed to modification: New comparison methods (e.g., text-based, HTML-based) can be added without changing the core logic. New comparison methods (e.g., text-based, HTML-based) can be added without changing the core logic. This is achieved using
  the ComparisonStrategy interface, which allows for plugging in different strategies without modifying WebsiteMonitorService.

By following these conventions, the resulting codebase remains clean, consistent, and easy to maintain, both by myself and others who might work on it in the future.