Design Patterns Analysis and Compliance Checks

INTRODUCTION

This document provides an analysis of the design patterns used in the development of the tire procurement dashboard and the compliance checks conducted to ensure data accuracy, security, and legal adherence. The use of established design patterns and thorough compliance checks were critical to the project's success, ensuring a robust, maintainable, and legally compliant system.

DESIGN PATTERNS ANALYSIS

1. MVC (MODEL-VIEW-CONTROLLER) PATTERN

- **Description:** The MVC pattern was used to separate the application's concerns, enhancing modularity and maintainability.
- **Implementation:**
  - **Model:** Represents the data structure, including the database schema for tire data, suppliers, and user information.
  - **View:** The user interface developed using Flutter, displaying data and capturing user input.
  - **Controller:** Handles the logic, processing user inputs from the view, and interacting with the model to fetch or update data. This includes the Flask application that serves as the backend API for data retrieval and processing.
- **Benefits:**
  - Improved code organization and separation of concerns.
  - Easier to maintain and scale the application.
- **Example:**
  - The tire price data is fetched from the database (Model), processed by the controller (Flask), and displayed on the dashboard (View).

2. SINGLETON PATTERN

- **Description:** Ensures a class has only one instance and provides a global point of access to it.
- **Implementation:**
  - Used for managing database connections and configurations within the Flask application.
  - A single instance of the database connection is created and reused throughout the application to avoid connection overhead and ensure consistent access.
- **Benefits:**
  - Reduced resource consumption by reusing a single instance.
  - Simplified access to shared resources.
- **Example:**
  - The database connection handler in the Flask application uses the Singleton pattern to manage a single connection instance for all database interactions.

3. OBSERVER PATTERN

- **Description:** Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **Implementation:**
  - Implemented to manage real-time updates on the dashboard.
  - When new tire data is scraped or existing data is updated, all relevant views (Flutter application) are notified and refreshed automatically.
- **Benefits:**

    - Ensures the dashboard displays the most up-to-date information without requiring manual refreshes.
    - Decouples data processing and UI updates.
- **Example:**
    - When new tire prices are fetched using Selenium and processed via Flask, the dashboard view in Flutter is automatically updated to reflect the latest prices.

4. STRATEGY PATTERN

- **Description:** Defines a family of algorithms, encapsulates each one, and makes them interchangeable. The strategy pattern lets the algorithm vary independently from clients that use it.
- **Implementation:**
    - Used for different data scraping techniques with Selenium.
    - Each scraping method is encapsulated in a separate class, allowing easy swapping or addition of new scraping methods.
- **Benefits:**
    - Flexibility to switch between different scraping algorithms without changing the core logic.
    - Simplified maintenance and enhancement of scraping functionalities.
- **Example:**
    - Switching between different Selenium scripts for scraping various supplier websites based on the website structure.

COMPLIANCE CHECKS

1. DATA ACCURACY AND VALIDATION

- **Description:** Ensuring the accuracy and validity of the collected data is paramount.
- **Implementation:**
    - Data validation scripts within Flask ensure that the data retrieved from Selenium is accurate and complete before being processed in n8n for normalization.
    - Automated tests were run to validate data against predefined schemas and rules.
- **Techniques:**
    - Schema validation using tools like JSON Schema or XML Schema Definitions (XSD).
    - Data consistency checks to ensure no duplicate or conflicting entries.
- **Example:**
    - Validating that tire prices are within expected ranges and that all mandatory fields (e.g., tire size, manufacturer) are populated before processing in n8n.

2. SECURITY COMPLIANCE

- **Description:** Ensuring the security of data and the application itself.
- **Implementation:**
    - Secure authentication and authorization mechanisms were implemented in Flask to protect user data.
    - Regular security audits and vulnerability assessments were conducted.
- **Techniques:**
    - Using HTTPS for secure data transmission.
    - Implementing role-based access control (RBAC) in Flask to restrict access to sensitive data.
    - Regularly updating dependencies to patch known vulnerabilities.
- **Example:**
    - User passwords are stored using strong hashing algorithms, and access to the dashboard is restricted based on user roles.

3. LEGAL AND ETHICAL COMPLIANCE

- **Description:** Ensuring that all data collection activities comply with legal and ethical standards.
- **Implementation:**
    - Explicit permissions were obtained from all suppliers before scraping their data using Selenium. Stakeholders requested approval from suppliers before starting the data collection process.
    - Adherence to robots.txt files and other website-specific scraping policies.
- **Techniques:**
    - Regular reviews of relevant legal regulations (e.g., GDPR, CCPA) to ensure compliance.
    - Maintaining detailed logs of data collection activities for audit purposes.
- **Example:**
    - Before scraping data from a supplier's website, the robots.txt file is checked to ensure compliance with the site's scraping policies, and written consent is obtained from the supplier.