**Report: Implementation of the Builder Pattern in Java with Clean Code Principles**

**Project Overview**

**Car.Builder — Builder Pattern implementation in Java**

**Author: Seidigali Rakhman**

**Group: SE-2411**

**Purpose:** This report describes the design, implementation, testing, and justification of a Java project that implements the Builder design pattern to construct Car objects. The implementation follows Clean Code principles and includes a Director for reusable build recipes.

**1. Abstract**

The Builder pattern is used to construct complex objects step-by-step and to decouple construction logic from the final representation. This project demonstrates a robust, maintainable implementation of the Builder pattern for a Car class. The implementation emphasizes Clean Code: small single-purpose methods, meaningful names, validation, immutability of the product, and use of interfaces to separate contract from implementation.

**2. Requirements & Acceptance Criteria**

**Functional requirements:**

* Provide a Car class with fields commonly used to describe vehicles (brand, model, year, mileage, color, engineType, isElectric, doors, transmission, price).
* Implement a nested CarBuilder that provides fluent configuration methods and a final build().
* Provide a CarBuilderInterface that defines builder methods to enable alternative builders in future.
* Provide a CarDirector that creates pre-configured cars (basic and luxury) using the builder interface.
* Include validation in builder methods and overall build() to prevent invalid configurations (e.g., electric cars must have engineType "Electric").

**Non-functional requirements:**

* Code should be readable, maintainable and follow Clean Code principles.
* The application must compile and run on Java 17+.
* Provide a clear report and instructions for running and submitting to GitHub.

**3. Design & Rationale**

**Why Builder?**

* The Car object contains many optional or configurable fields. Using many constructors (telescoping constructors) or a long constructor with multiple parameters harms readability and increases error risk. Setters after construction can lead to partially initialized objects.
* Builder provides: readable fluent API, validation at construction time, immutability of the final object, and separation of construction logic.

**Class responsibilities**

* Car (Product): immutable once built, exposes getters and toString() for presentation.
* Car.CarBuilder (Builder): holds intermediate state, provides validation, and creates Car via private constructor.
* CarBuilderInterface: contract for builder methods to allow swapping builder implementations.
* CarDirector (Director): exposes common build recipes to quickly create typical configurations.
* Main: demonstration and manual tests.

**Clean Code highlights**

* **Small methods**: Each setter method performs small validation + assignment.
* **Descriptive names**: setBrand, constructLuxuryCar, CarBuilderInterface.
* **Validate early**: builder methods throw IllegalArgumentException for invalid inputs.
* **Immutability**: Car has only private final-like fields (set only in constructor) and no setters.
* **Single responsibility**: Builder handles construction, Car handles data storage.