**TP-1: CleaningData - Applying OOP and Design Patterns**

Each exercise builds upon the previous one - so by the end, we will have a **complete CSV cleaning data** using **OOP, decorators, and patterns**.

**Exercise 1: Build the CSVReader Class (OOP Foundation)**

**Objective:**

* Reinforce **OOP fundamentals**: classes, constructors, methods, encapsulation.
* Learn to design reusable data classes for DS projects.

**Concept:**

A CSVReader class should encapsulate all functionality related to reading CSV files.

**Instructions:**

1. Create a class CSVReader.
2. Define an \_\_init\_\_ method that takes file\_path.
3. Add a read() method that loads the CSV file using pandas.
4. Add a preview(n) method that prints the top n rows.

**Exercise 2: Apply Pattern for Missing Values Handling**

**Objective:**

* Learn the **Strategy Pattern** - one interface, many interchangeable behaviors.
* Practice **inheritance** and **polymorphism**.

**Concept:**

We often need different strategies to handle missing data (drop, fill mean, fill mode, etc.).  
The pattern allows flexible switching between methods **without changing the main code**.

**Instructions:**

1. Create an abstract class MissingValueStrategy with a method handle(df).
2. Create subclasses:
   * DropMissing
   * FillMean
   * FillMode
3. Create a DataCleaner class that applies the chosen strategy.

**Exercise 3: Add Decorators for Logging and Timing**

**Objective:**

* Learn the **Decorator Pattern** to add reusable functionality (logging, timing).
* Understand how decorators support the “Open/Closed Principle”.

**Concept:**

Decorators wrap functions to extend their behavior without altering the original code.

**Instructions:**

1. Create two decorators:
   * @log\_action → logs when a method starts and finishes.
   * @log\_time → calculates and prints execution time.
2. Apply them to methods in CSVReader or DataCleaner.

**Exercise 4: Implement Factory Pattern for Data Transformations**

**Objective:**

* Practice **Factory Pattern** for scalable creation of transformation objects.
* Apply **abstraction** and **composition**.

**Concept:**

Instead of manually creating objects, use a **factory** that decides what transformation to apply.

**Instructions:**

1. Create an abstract class DataTransform with a method apply(df).
2. Implement subclasses:
   * NormalizeColumns
   * RemoveDuplicates
   * StandardizeText
3. Create a TransformFactory that returns transformation objects based on string input.

**Exercise 5: Build a Full Cleaning Pipeline (Template Method Pattern)**

**Objective:**

* Combine all previous concepts into one **cleaning pipeline**.
* Use the **Template Method Pattern** to define a consistent workflow.

**Concept:**

The Template Method defines the skeleton of a process and lets subclasses override specific steps.

**Instructions:**

1. Create an abstract class DataPipeline with a run() method defining steps:
   * load()
   * clean()
   * transform()
   * save()
2. Create CSVDataPipeline that implements each step using previous classes.