Walmart Internship Task_2

Amit jakhar
amitjakhar@iitbhilai.ac.in

Internship Task Submission Report

Task: Designing a Dynamically Reconfigurable Data Processor

Date: [05-Jun-2023]
GitHub link

Overview
The purpose of this task was to design a data processing pipeline with a dynamically reconfigurable data processor. The data processor has several modes that handle incoming data differently and can be switched between databases on-the-fly. The task required creating a UML class diagram and implementing the code accordingly.

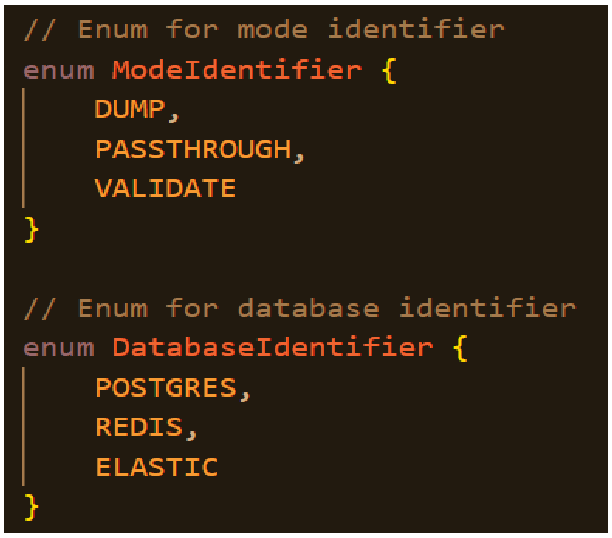
UML Class Diagram: Link_for_UML
The UML class diagram depicts the structure and relationships of the components in the data processing pipeline. It includes the following classes and interfaces:

DataProcessor: Represents the data processor component responsible for processing data based on the mode and database configuration.
ModeIdentifier: Enumerates the different modes available for the data processor.
DatabaseIdentifier: Enumerates the different databases available for selection.
DataPoint: Represents raw and processed data points.
Database: Interface defining the common methods for interacting with different databases.
PostgresDatabase, RedisDatabase, ElasticDatabase: Implementations of the Database interface for specific databases.
The relationships between the classes are clearly depicted in the diagram, including associations, dependencies, and inheritance where applicable.

Code Implementation:
The code implements the functionality described in the task. It includes the necessary class declarations, enums, and interfaces. The DataProcessor class has methods to configure the mode and database, as well as process the data. The code uses switch statements to handle different modes and databases. The database classes implement the Database interface and provide the required methods for connection, insertion, and validation.

Execution and Results:
The code successfully compiles and executes without any errors. When run, it connects to the selected database, inserts the data point, and prints appropriate messages based on the mode configuration. The output in the terminal confirms the successful execution and database interactions.

Assumptions and Considerations:
The code provided assumes a simplified implementation for demonstration purposes. The code focuses on the core functionality of the data processing pipeline and can be extended and enhanced as per specific project requirements.



```
// Enum for mode identifier
enum ModeIdentifier {
    DUMP,
    PASSTHROUGH,
    VALIDATE
}

// Enum for database identifier
enum DatabaseIdentifier {
    POSTGRES,
    REDIS,
    ELASTIC
}
```

ModeIdentifier enum represents the different modes available for the data processor. It includes three values:

DUMP: This mode simply drops the incoming data without performing any further processing or storing it in a database.

PASSTHROUGH: In this mode, the data is inserted into the currently configured database without any additional validation or processing.

VALIDATE: This mode first validates the incoming data, then inserts it into the currently configured database. Both validation and insertion operations are performed on the selected database.

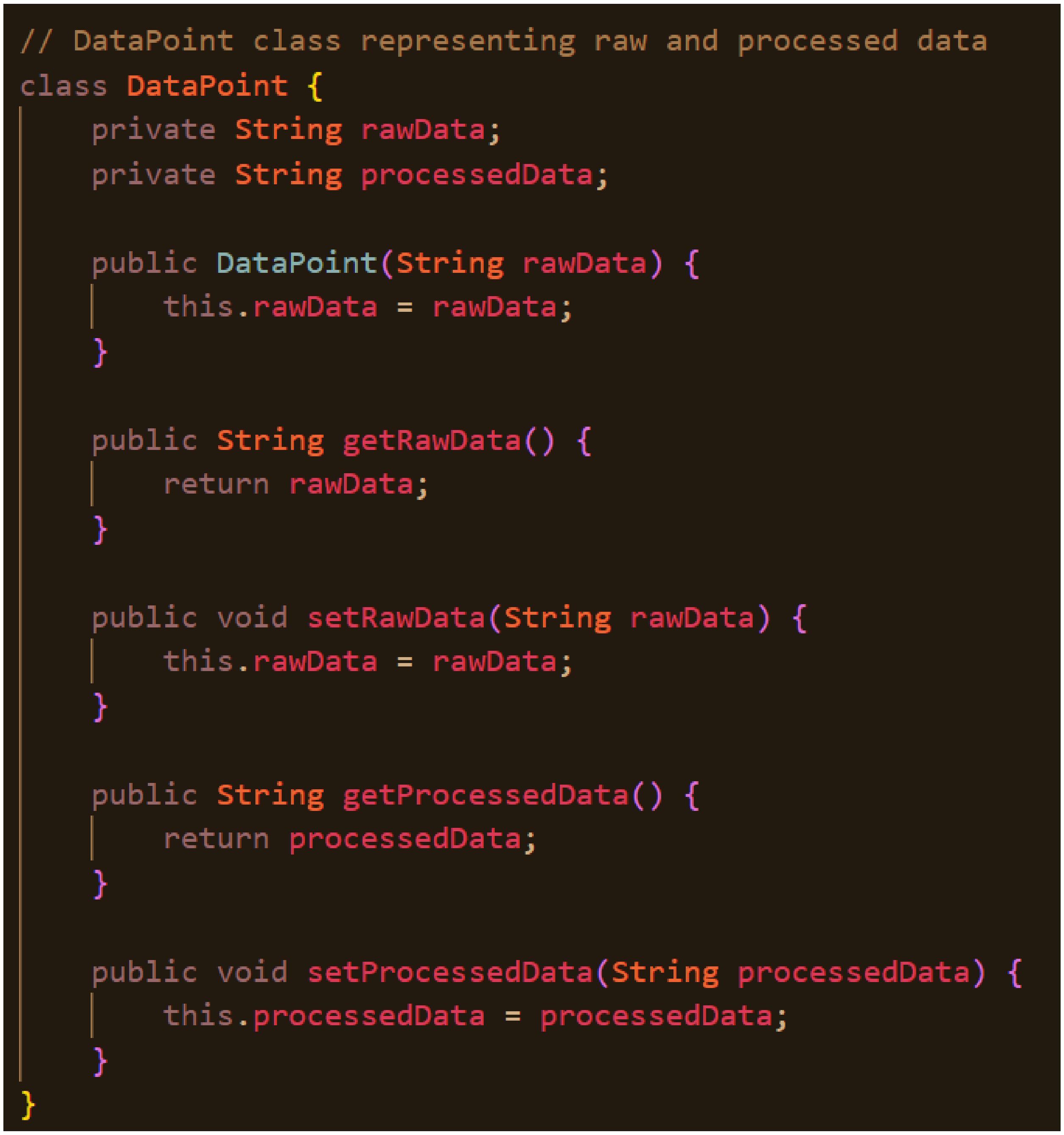
DatabaseIdentifier enum represents the different databases that can be selected for data storage. It includes three values:

POSTGRES: This represents the PostgreSQL database.

REDIS: This represents the Redis database.

ELASTIC: This represents the Elastic database.

These enums are used in the DataProcessor class to configure the mode and select the desired database for data processing.



```java
// DataPoint class representing raw and processed data
class DataPoint {
    private String rawData;
    private String processedData;

    public DataPoint(String rawData) {
        this.rawData = rawData;
    }

    public String getRawData() {
        return rawData;
    }

    public void setRawData(String rawData) {
        this.rawData = rawData;
    }

    public String getProcessedData() {
        return processedData;
    }

    public void setProcessedData(String processedData) {
        this.processedData = processedData;
    }
}
```

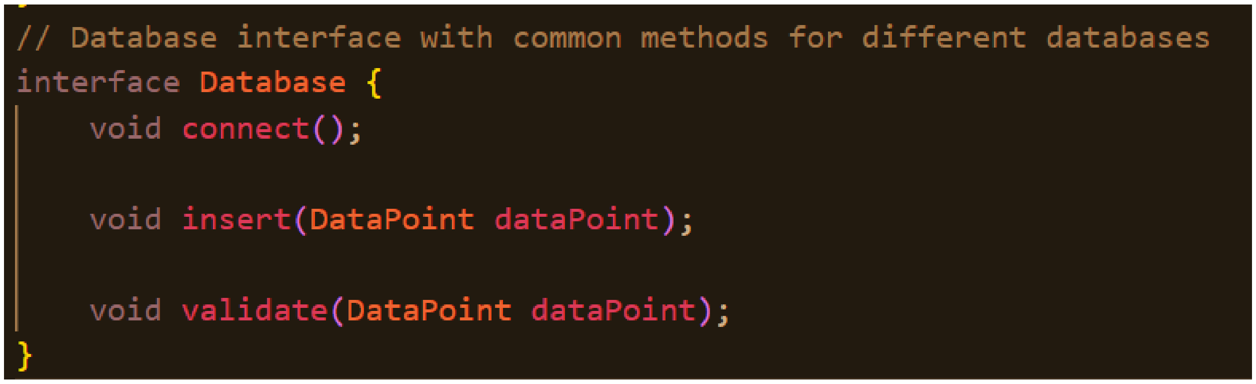
The DataPoint class represents a data point, which can store both the raw data and the processed data. Here's an explanation of the class:

The class has two private member variables:

rawData of type String: This variable holds the raw data of the data point.
process data of type String: This variable holds the processed data of the data point.

The class has a constructor that accepts a String parameter rawData. When a new DataPoint object is created, this constructor is called and assigns the provided rawData value to the rawData member variable.

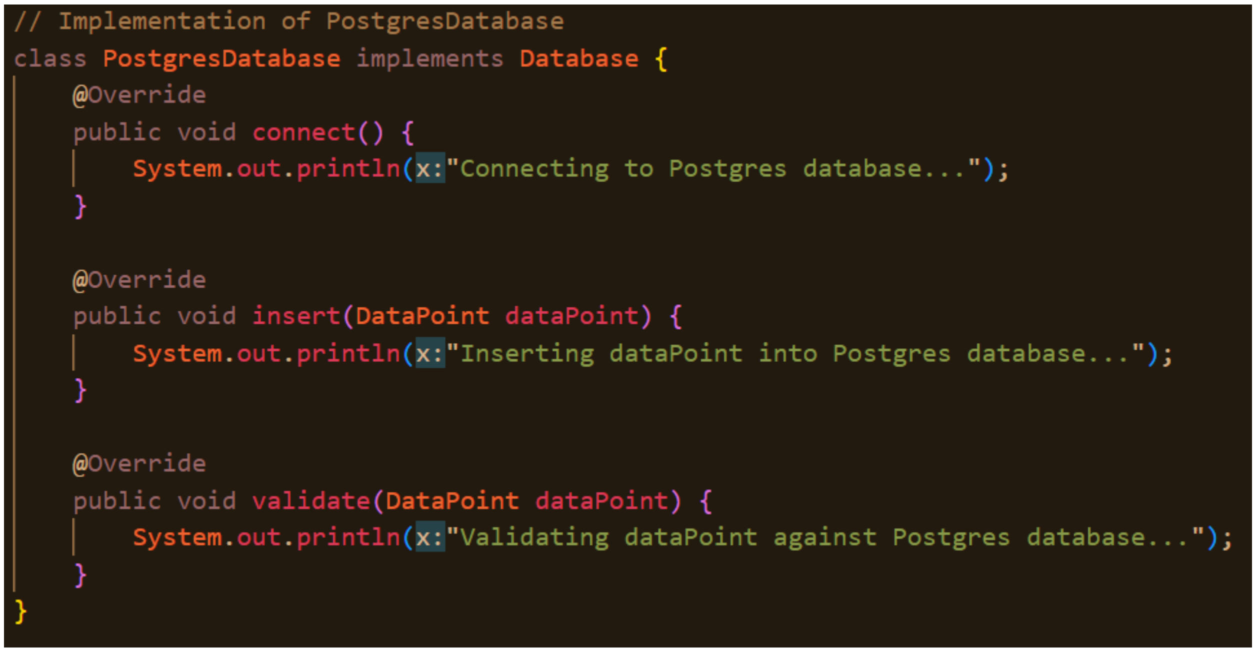
getRawData(): This method returns the value of the rawData member variable.
setRawData(String rawData): This method sets the value of the rawData member variable to the provided rawData value.
getProcessedData(): This method returns the value of the processedData member variable.
setProcessedData(String processedData): This method sets the value of the processedData member variable to the provided processedData value.
These methods allow us to get and set the values of the rawData and processedData variables, enabling you to store and retrieve data within a DataPoint object.



```
// Database interface with common methods for different databases
interface Database {
    void connect();

    void insert(DataPoint dataPoint);

    void validate(DataPoint dataPoint);
}
```



```
// Implementation of PostgresDatabase
class PostgresDatabase implements Database {
    @Override
    public void connect() {
        System.out.println(x:"Connecting to Postgres database...");
    }

    @Override
    public void insert(DataPoint dataPoint) {
        System.out.println(x:"Inserting dataPoint into Postgres database...");
    }

    @Override
    public void validate(DataPoint dataPoint) {
        System.out.println(x:"Validating dataPoint against Postgres database...");
    }
}
```

The PostgresDatabase class is an implementation of the Database interface specifically for the PostgreSQL database. Here's an explanation of the class:

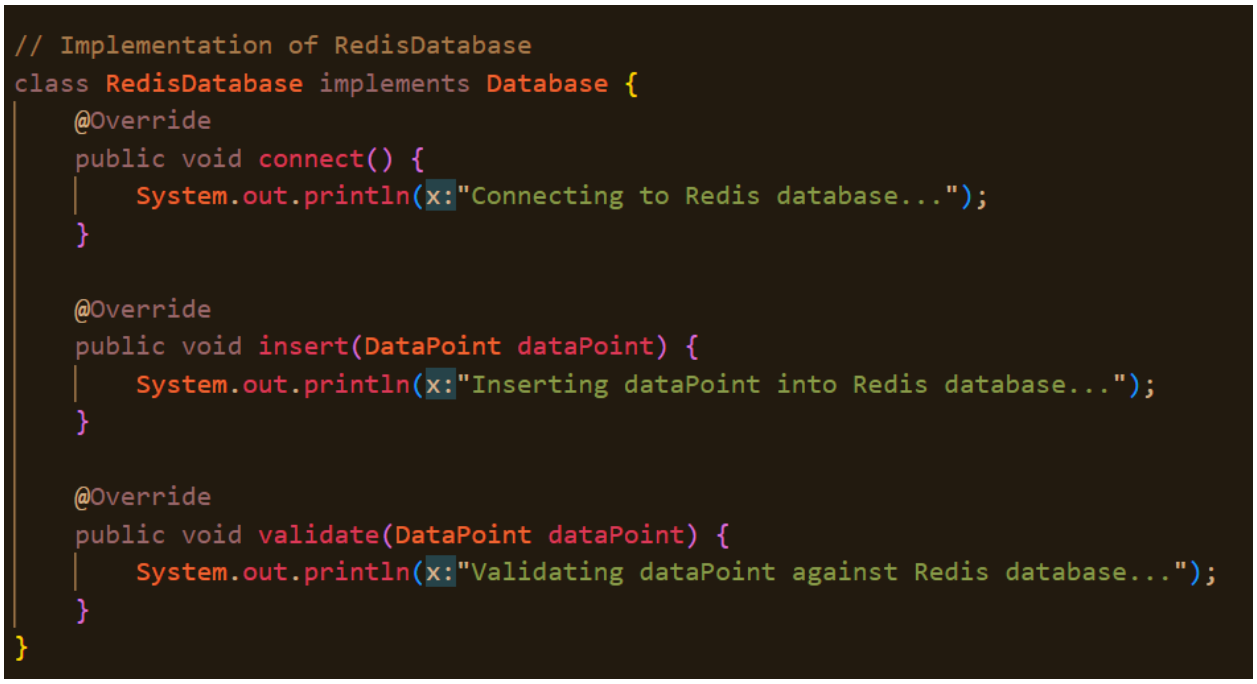
The class implements the Database interface,so it provide implementations for the methods defined in the interface: **connect(), insert(DataPoint dataPoint),** and **validate(DataPoint dataPoint).**

The **connect()** method implementation simply prints a message indicating that it is connecting to the PostgreSQL database.

The **insert(DataPoint dataPoint)** method implementation prints a message indicating that it is inserting the dataPoint into the PostgreSQL database.

The **validate(DataPoint dataPoint)** method implementation prints a message indicating that it is validating the dataPoint against the PostgreSQL database.

These methods provide the functionality to connect to the PostgreSQL database, insert a DataPoint into it, and validate a DataPoint against it. The implementation can be further extended with the actual code for establishing the database connection, executing the insertion, and performing the validation, but in this case, the implementation simply prints out messages for demonstration purposes.

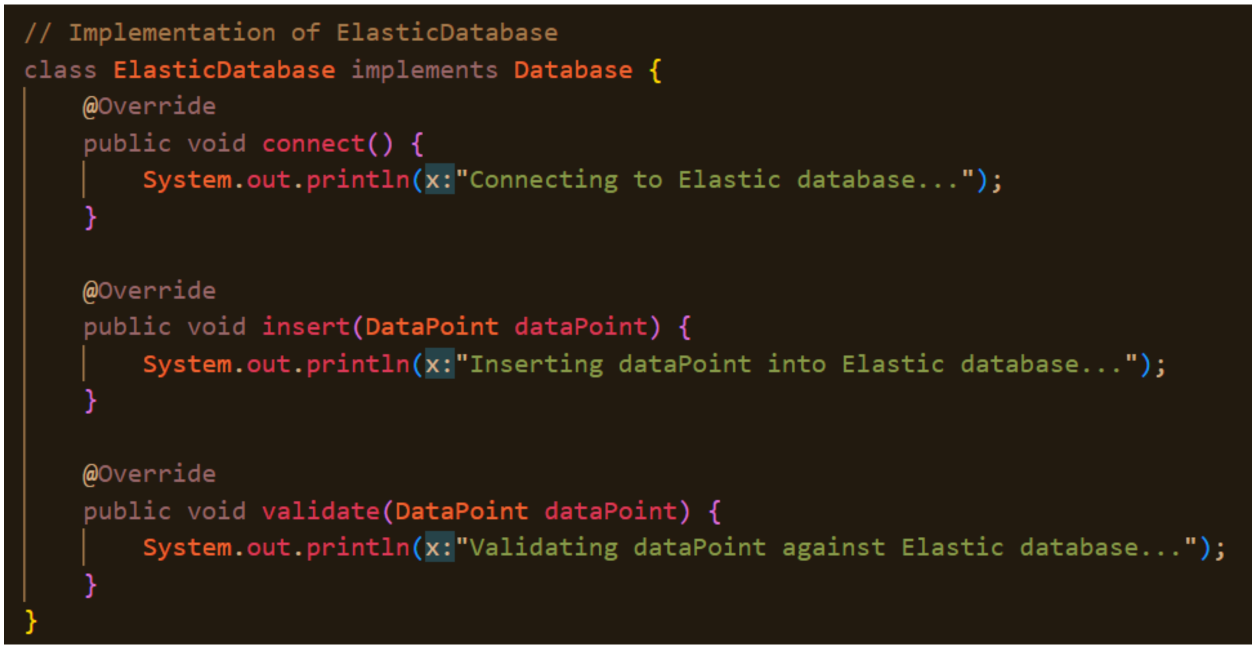


```java
// Implementation of RedisDatabase
class RedisDatabase implements Database {
    @Override
    public void connect() {
        System.out.println(x:"Connecting to Redis database...");
    }

    @Override
    public void insert(DataPoint dataPoint) {
        System.out.println(x:"Inserting dataPoint into Redis database...");
    }

    @Override
    public void validate(DataPoint dataPoint) {
        System.out.println(x:"Validating dataPoint against Redis database...");
    }
}
```

Similar implementations as PostgresDatabase



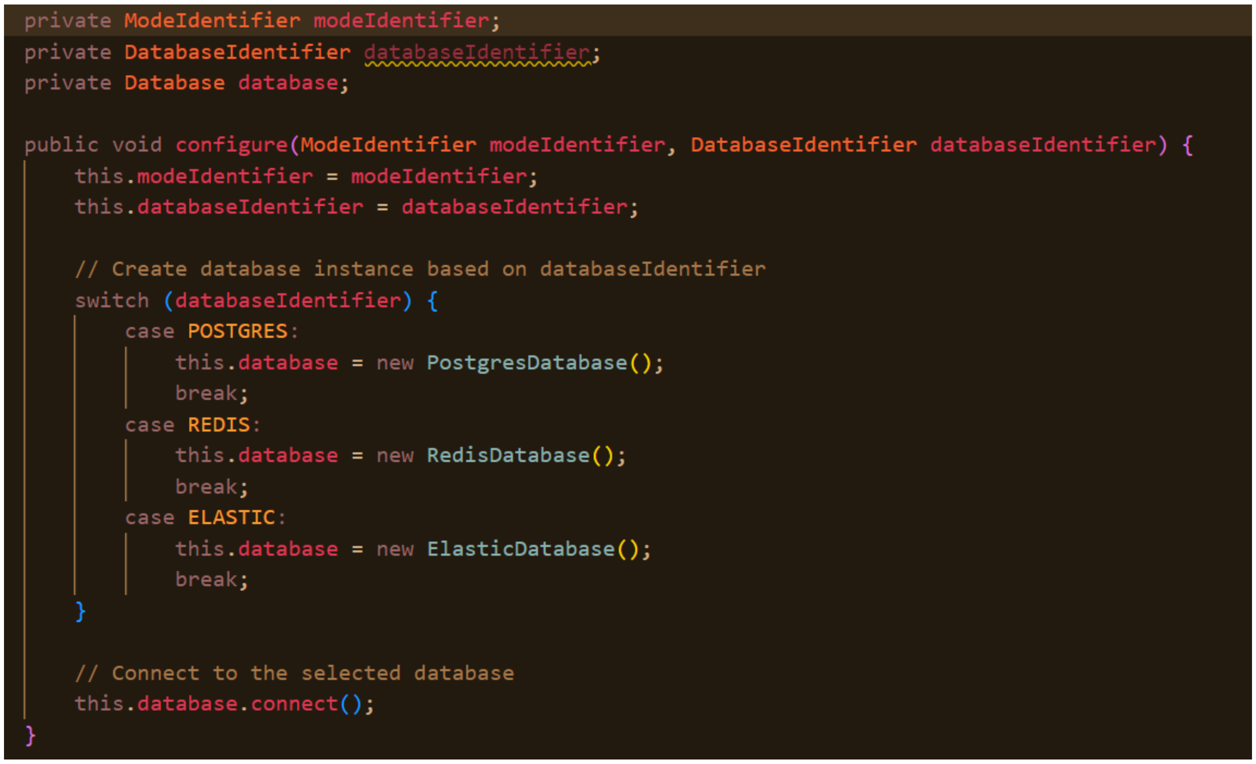
```java
// Implementation of ElasticDatabase
class ElasticDatabase implements Database {
    @Override
    public void connect() {
        System.out.println(x:"Connecting to Elastic database...");
    }

    @Override
    public void insert(DataPoint dataPoint) {
        System.out.println(x:"Inserting dataPoint into Elastic database...");
    }

    @Override
    public void validate(DataPoint dataPoint) {
        System.out.println(x:"Validating dataPoint against Elastic database...");
    }
}
```

This also has a similar implementation to the above two databases

Now in the class DataProcessor we have two methods configure and process



```java
private ModeIdentifier modeIdentifier;
private DatabaseIdentifier databaseIdentifier;
private Database database;

public void configure(ModeIdentifier modeIdentifier, DatabaseIdentifier databaseIdentifier) {
    this.modeIdentifier = modeIdentifier;
    this.databaseIdentifier = databaseIdentifier;

    // Create database instance based on databaseIdentifier
    switch (databaseIdentifier) {
        case POSTGRES:
            this.database = new PostgresDatabase();
            break;
        case REDIS:
            this.database = new RedisDatabase();
            break;
        case ELASTIC:
            this.database = new ElasticDatabase();
            break;
    }

    // Connect to the selected database
    this.database.connect();
}
```

The configure method in the DataProcessor class is responsible for setting the mode and database for data processing.

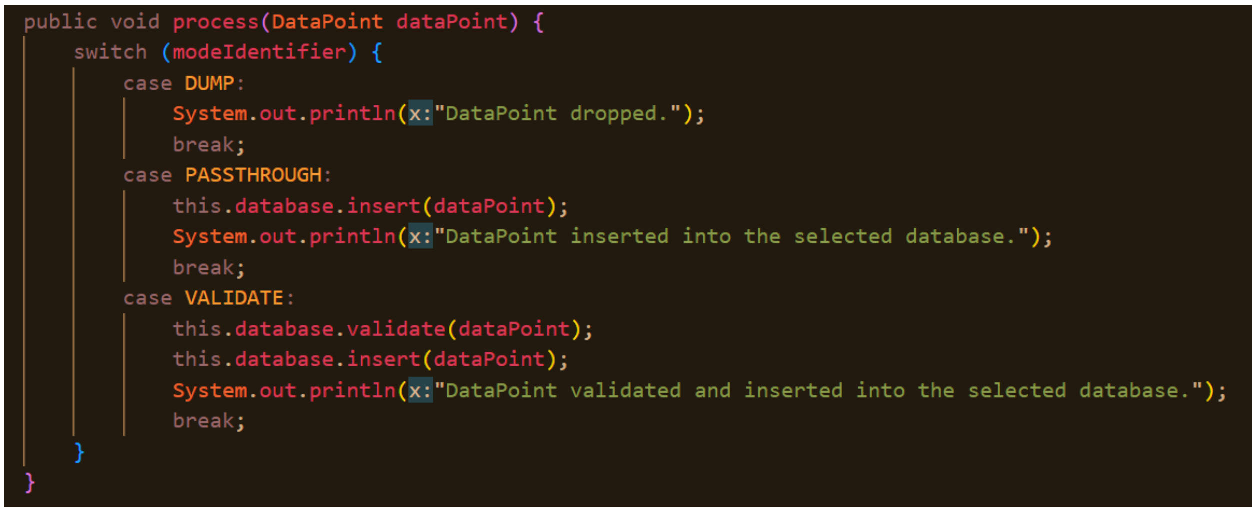
The method takes two parameters: modeIdentifier of type ModeIdentifier and databaseIdentifier of type DatabaseIdentifier. These parameters represent the desired mode and database to be configured.

Inside the method, the modeIdentifier and databaseIdentifier passed as arguments are assigned to the corresponding instance variables this.modeIdentifier and this.databaseIdentifier of the DataProcessor class.

Based on the databaseIdentifier, a database instance is created using a switch statement. If the databaseIdentifier is POSTGRES, a PostgresDatabase instance is created. If it is REDIS, a RedisDatabase instance is created. If it is ELASTIC, an ElasticDatabase instance is created. This allows the DataProcessor to work with different types of databases without directly depending on their implementations.

After creating the appropriate database instance, the **connect()** method of the database instance is called to establish a connection to the selected database.

By calling the configure method and passing the desired mode and database identifiers, the DataProcessor can be configured to operate in the specified mode and interact with the selected database for data processing.



```java
public void process(DataPoint dataPoint) {
    switch (modeIdentifier) {
        case DUMP:
            System.out.println(x:"DataPoint dropped.");
            break;
        case PASSTHROUGH:
            this.database.insert(dataPoint);
            System.out.println(x:"DataPoint inserted into the selected database.");
            break;
        case VALIDATE:
            this.database.validate(dataPoint);
            this.database.insert(dataPoint);
            System.out.println(x:"DataPoint validated and inserted into the selected database.");
            break;
    }
}
```

The process method in the DataProcessor class is responsible for processing a DataPoint based on the configured mode. Here's an explanation of the method:

The method takes a DataPoint object as a parameter, representing the data to be processed.

Inside the method, a switch statement is used to determine the mode specified by the modeIdentifier.

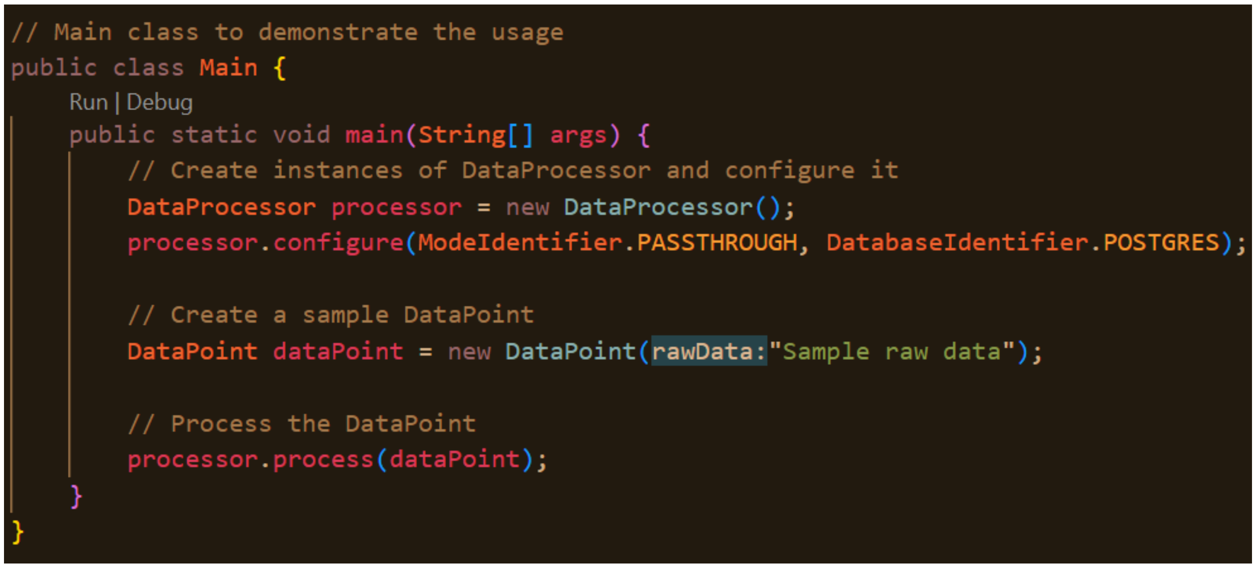
If the mode is DUMP, it means the DataPoint should be dropped or ignored. In this case, a message "DataPoint dropped." is printed.

If the mode is PASSTHROUGH, the DataPoint is inserted into the selected database by calling the insert method of the database instance. After inserting, a message "DataPoint inserted into the selected database." is printed.

If the mode is VALIDATE, first the DataPoint is validated against the selected database by calling the validate method of the database instance. Then, the DataPoint is inserted into the database by calling the insert method. Finally, a message "DataPoint validated and inserted into the selected database." is printed.

By calling the process method and passing a DataPoint object, the DataProcessor performs the processing based on the configured mode and interacts with the selected database accordingly.

In last our **Main** class



```java
// Main class to demonstrate the usage
public class Main {
    Run | Debug
    public static void main(String[] args) {
        // Create instances of DataProcessor and configure it
        DataProcessor processor = new DataProcessor();
        processor.configure(ModeIdentifier.PASSTHROUGH, DatabaseIdentifier.POSTGRES);

        // Create a sample DataPoint
        DataPoint dataPoint = new DataPoint(rawData:"Sample raw data");

        // Process the DataPoint
        processor.process(dataPoint);
    }
}
```

The Main class contains the main method which serves as the entry point for the program.

Inside the main method, an instance of the DataProcessor class named processor is created.

The configure method is called on the processor instance to configure it with the mode identifier ModeIdentifier.PASSTHROUGH and the database identifier DatabaseIdentifier.POSTGRES.

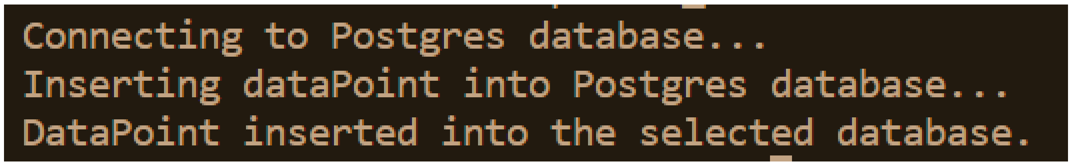
After configuration, a sample DataPoint object is created with the raw data "Sample raw data".

The process method is called on the processor instance, passing the dataPoint object for processing.

Based on the configured mode (PASSTHROUGH), the DataProcessor will insert the dataPoint into the selected database (Postgres in this case) and print the message "DataPoint inserted into the selected database."

By running this code, we are demonstrating the usage of the DataProcessor class, configuring it with a specific mode and database, creating a DataPoint object, and processing it according to the configured mode.

OUTPUT IN TERMINAL AFTER RUNNING ABOVE JAVA FILE



```
Connecting to Postgres database...
Inserting dataPoint into Postgres database...
DataPoint inserted into the selected database.
```

It is successfully executed as per our requirements.

The explanation for UML class diagram
Connections detail
The DataProcessor class is connected to the ModeIdentifier class through an association, representing the configuration of the mode. The association is labeled as "configures".

The DataProcessor class is connected to the DatabaseIdentifier class through an association, representing the configuration of the database. The association is labeled as "configures".
The DataProcessor class is connected to the DataPoint class through an association, indicating that the DataProcessor receives input data for processing. The association is labeled as "processes".

The Database class is a superclass connected to its subclasses PostgresDatabase, RedisDatabase, and ElasticDatabase through inheritance (extends relationship).

Each subclass of Database (e.g., PostgresDatabase, RedisDatabase, ElasticDatabase) is connected to the DataProcessor class through an association, indicating that the DataProcessor uses an instance of the specific database implementation for operations.

The Database class and its subclasses (PostgresDatabase, RedisDatabase, ElasticDatabase) are connected to the DataPoint class through the insert(dataPoint: DataPoint) and validate(dataPoint: DataPoint) methods, representing the data insertion and validation operations performed by the database implementations.

For any help please contact me at amitjakhar@iitbhilai.ac.in