

# Email Classifier System

CS4125 Systems Analysis and Design  
Design Document

Kevin Collins 21344256

Olan Healy 21318204

Brendan Quinn 21315205

Caoimhe Cahill 21331308



# Table Of Contents

Table Of Contents .....	2
Project Overview .....	3
Objectives .....	3
Use Cases .....	4
Functional Requirements.....	5
Non-Functional Requirements .....	6
Preprocessing Workflow Overview.....	6
1. Text Cleaning (text_cleaner.py).....	6
2. Translation (translator.py).....	7
3. Data Preparation and Labelling (preprocessor.py) .....	7
4. Preprocessing Execution (preprocess.py).....	8
Rationale for File-Based Modularity.....	9
Design Patterns.....	9
Factory Pattern .....	9
Purpose .....	9
Why we choose factory pattern .....	9
How the factory pattern is used.....	10
Strategy Pattern .....	12
Purpose .....	12
Why we choose strategy .....	12
How the strategy pattern is used .....	13
Singleton Pattern .....	16
Purpose .....	16
Why we chose the Singleton Pattern.....	17

How singleton pattern is used .....	17
Observer Pattern.....	21
Purpose .....	21
Why we choose the Observer Pattern .....	21
How the Observer Pattern is used .....	22
Decorator Pattern .....	24
Purpose .....	24
Why we choose the Decorator Pattern.....	24
How the Decorator Pattern is used .....	25
Email Classifiers Implemented .....	30
Sequence Diagram .....	30
State Diagram.....	32
Use Of Agile For collaboration on our project.....	32

## Project Overview

This project is an Email Classifier which is designed to automatically categorize emails into categories:

The purpose of the project is to be built using a modular design, so different machine learning models and design patterns can be implemented to allow flexibility and adaptability. Users can select and switch between different machine learning algorithms for classification dynamically.

## Objectives

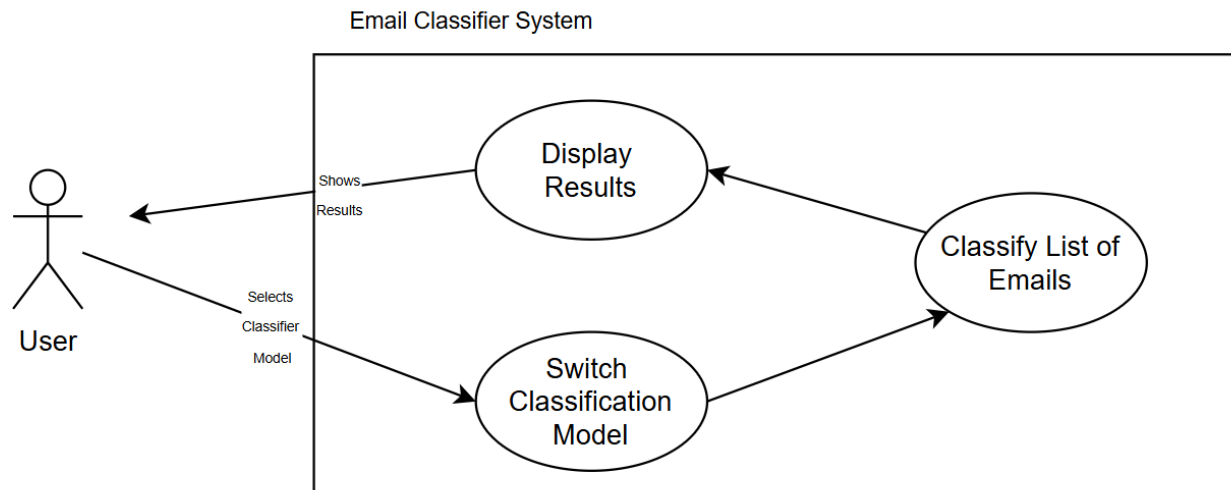
Our goal was to develop a modular and extendible system which is capable of automatically classifying emails into categories. Our system will leverage Natural Language Processing models along with heuristics to deliver an accurate classification of a list of emails.

In this project, although we will attempt to maintain the accuracy and performance of the model, our main focus is on the system's architectural design. We will harness various design patterns to create a flexible software architecture. Through this approach we allow

our system to adapt to new requirements which may arise. This allows for new requirements to be implemented with minimal restructuring.

The design patterns we apply will address different concerns such as data cleaning, model selection and seamless classification.

## Use Cases



### USE CASE ONE: Classify a list of emails

**Goal in context:** The system automatically classifies an incoming email into one of the predefined categories based on the Natural Language Processing Model and heuristic rules

**Actors:** User, System

**Preconditions:** The email is received by the system where classification model and heuristics should be set up.

#### Main Flow:

1. The system allows the user to select the model they wish to classify the data with.
2. Email list is received by the system.
3. The system performs preprocesses to verify cleanliness of the data.
4. The selected model classifies the email list into predefined categories.
5. The classification results are displayed.

**Postconditions:** The list of emails is successfully classified into the correct category and the results are printed.

## **USE CASE TWO:** Switch Classification Model

**Goal in context:** Allow the user to select the classification model to be used.

**Actors:** User, System

**Preconditions:** The system has multiple classification models implemented and are available for selection.

### **Main Flow:**

1. The user selects a classification model from the available options.
2. The system applies the selected model configuration.
3. The system performs preprocesses to verify cleanliness of the data.
4. The selected model classifies the email list into predefined categories.
5. The classification results are displayed.

**Postconditions:** The selected model is successfully configured for use.

## **USE CASE THREE:** View Classification Results

**Goal in context:** The system should display the classification results.

**Actors:** User, System

**Preconditions:** The system has successfully classified the email dataset.

### **Main Flow:**

1. The selected model classifies the email list into categories.
2. The classification results are displayed.

**Postconditions:** Classification results are displayed to the user.

# Functional Requirements

**Preprocessing Logic:** The system must allow for preprocessing of emails to clean the data to allow our model to perform seamless classification.

**Dynamic Model Selection Mechanism:** A flexible component must allow for using different Classification models.

**Classification Logic:** The functionality allows our system to classify emails on predefined categories using both Natural Language Processing models and heuristic methods.

**Classification Rules:** A set of heuristic rules which a classification model will use to classify an email.

**Accuracy Reporting:** The system should provide metrics of its classification for each category.

**Pattern-Driven Architecture:** Allow for flexible configuration if additional requirements arise.

**Logging Functionality:** The system should log each classification result and provide a summary of when all data is classified.

## Non-Functional Requirements

**Modularity:** The system should employ design patterns to ensure that the systems components are loosely coupled allowing for easy maintenance and enhancements.

**Performance:** The application should be able to classify emails efficiently to ensure high throughput.

**User Documentation:** Documentation is needed to explain to the user how to operate the system.

## Preprocessing Workflow Overview

The preprocessing pipeline for this project efficiently prepares raw text data by cleaning, translating, and structuring it into labelled datasets suitable for machine learning.

### *1. Text Cleaning (text\_cleaner.py)*

The first step in preprocessing is cleaning the raw text data to ensure it is free from noise and irrelevant patterns. The `clean_text` function plays a role in this process.

#### **Functionality:**

- **Pattern Removal:**
  - Removes common noise such as email signatures (e.g., "Sent from my..."), placeholders like "Customer Support," and reply prefixes like "Re:".
- **Special Character Filtering:**

- Eliminates unnecessary single-character symbols and extra spaces, leaving the text concise and standardised.
- **String Validation:**
  - Ensures non-string entries are converted to empty strings, maintaining consistency in the dataset.

By applying these cleaning rules, the dataset is significantly simplified, retaining only meaningful content. This step enhances downstream processes like translation and labelling by reducing irrelevant variations in the data.

## *2. Translation (translator.py)*

The cleaned text often includes multilingual entries, which can introduce variability that affects modeling. The `translate_to_english` function ensures that all text is converted into English.

### **Functionality:**

- **Translation:**
  - Uses Google Translate to convert non-English text into English, standardizing the language for all entries.
- **Error Tracking:**
  - Maintains counters for successful and failed translations, enabling detailed reporting on translation success rates.
- **Translation Reporting:**
  - The `report_translation_rate` function calculates the percentage of successfully translated lines and prints this information for monitoring and debugging. We reach around 90% success usually due to the limitations of google translate and the database.

## *3. Data Preparation and Labelling (preprocessor.py)*

This file integrates cleaning and translation and adds hierarchical labels and preparing the data for training and evaluation.

### **Functionality:**

- **Column Selection:**
  - Retains only relevant columns, such as Interaction content, Type 1, Type 2, etc., for processing.
- **Data Cleaning and Translation:**
  - Sequentially applies `clean_text` and `translate_to_english` to standardize text content.
- **Hierarchical Labelling:**
  - Combines nested categories (Type 1, Type 2, Type 3, and Type 4) into a single hierarchical label (Type 1 > Type 2 > Type 3 > Type 4), simplifying the classification problem while preserving relationships between levels.
- **Missing Value Handling:**
  - Replaces missing values with "Unknown" ensuring the dataset remains consistent and complete.

The resulting dataset is saved as `data/preprocessed_appgallery_data_with_types.csv` for example. This step ensures the data is not only clean and translated but also structured with meaningful labels.

#### *4. Preprocessing Execution (preprocess.py)*

The `preprocess.py` script acts as the orchestrator for running the entire pipeline. It demonstrates the integration of the modular components.

#### **Functionality:**

- **Data Loading:**
  - Reads the raw dataset from a CSV file into a DataFrame.
- **Pipeline Execution:**
  - Calls `preprocess_data_with_splits` from `preprocessor.py` to clean, translate, and label the data.
- **Output Handling:**
  - Saves the preprocessed data to a CSV file (`data/preprocessed_appgallery_data_with_types.csv`), making it reusable for training and testing scripts.



This script provides a simple interface for running the entire preprocessing pipeline, allowing users to process raw datasets efficiently.

### *Rationale for File-Based Modularity*

#### 1. **Separation of Concerns:**

- Each file handles a specific aspect of preprocessing, such as cleaning, translating, or structuring, ensuring maintainability and clarity.

#### 2. **Reusability:**

- Individual components like `clean_text` or `translate_to_english` can be reused in other projects or workflows.

#### 3. **Ease of Debugging:**

- Modular design makes it easy to isolate and address issues in specific stages of preprocessing.

#### 4. **Scalability:**

- Adding new preprocessing steps or modifying existing ones is straightforward due to the clear division of responsibilities across files.

## Design Patterns

### Factory Pattern

#### *Purpose*

We used the factory design pattern to provide a centralised mechanism to create and configure each of our machine learning models for email classification. This allows the dynamic creation of different classifiers based on our configuration. This design pattern abstracts instantiation logic which ensures that the user can classify their email without needing to know how it's configured or its details. This modular approach ensures scalability, flexibility, and adheres to single responsibility principle.

#### *Why we choose factory pattern*

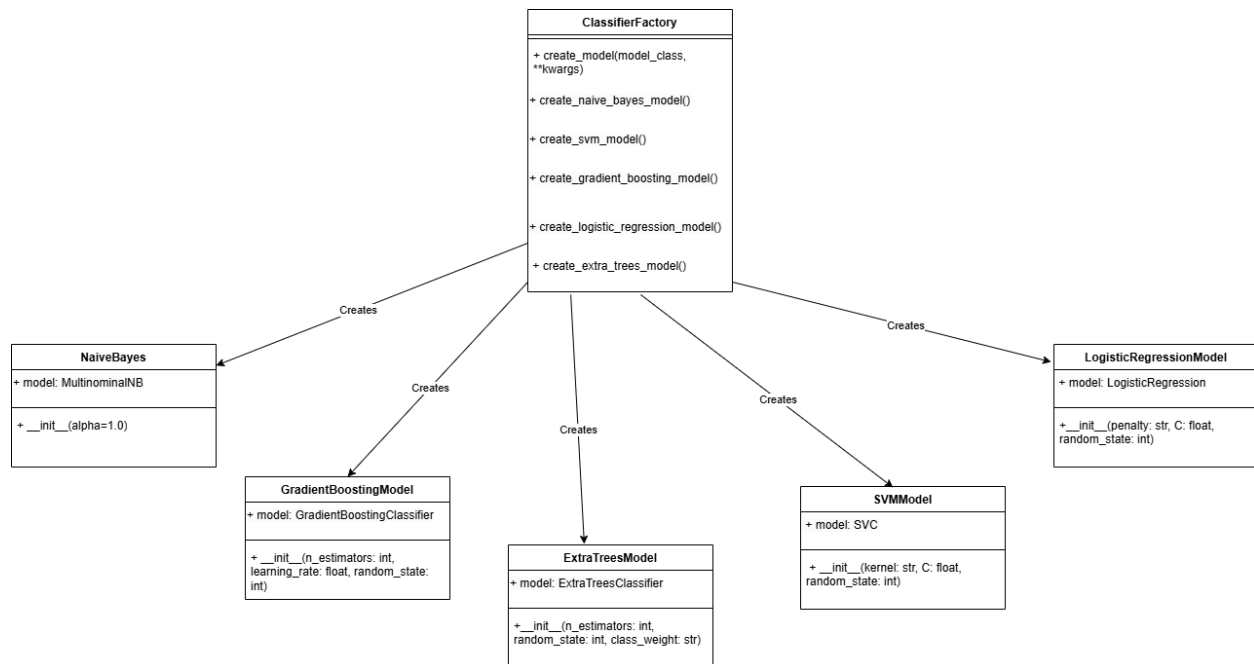
1. **Simplifies model Creation:** It consolidates the creation of multiple machine learning classifiers into a single, well-defined structure, avoiding redundant code.

2. **Dynamic Configuration:** As it integrates with our singleton configuration class, it supports parameterised model creation, which enables on the fly adjustments without modifying the core code
3. **Encapsulation of complex logic:** Say if a non-programmer were to use this classifier, they might not understand what learning rate is etc. This process is hidden.
4. **Potential for future expansion:** Adding a new classifier would be a straightforward process by just adding a new method to the factory class, eg RandomTrees from the lab.
5. **Modularity:** By decoupling our model creation from the core logic, it makes our design clean and maintainable

#### *How the factory pattern is used*

The 'ClassifierFactory' class provides static and class methods to create the different models for classification with unique configurations. The classifiers are created dynamically and returned to the main classification pipeline for training and predictions to take place





## Strategy Pattern

### Purpose

The strategy pattern is used in our system to encapsulate the behaviors of our different models created via factory pattern. We defined a common interface to ensure the strategy pattern can be used seamlessly and dynamically within our system. This approach ensures flexibility as it allows the switching of algorithms during runtime based on requirements or needs of the user. It also encapsulates behavior for each algorithm, which makes the system modular and easy to extend. It also supports the addition of new algorithms (along with factory) without modifying existing strategies.

### Why we choose strategy

1. **Encapsulation of behavior:** Each class encapsulates the specific behavior of a classification algorithm, ensuring the logic is reusable and isolated.
2. **Dynamic Algorithm selection:** Supports dynamic switching between different algorithms during runtime, so if we were to have different datasets you could choose a more suited algorithm.
3. **Ease of extensibility:** Adding a new algorithm only involves creating a new strategy class implementing the ClassificationStrategy class (once model is added to factory). This ensures that the existing code remains untouched, supporting open closed principle.
4. **Modularity:** Decouples the behavior from model instantiation, making the system clean and easy to debug.

### *How the strategy pattern is used*

Firstly, we define an abstract base class called 'ClassificationStrategy'. This defines a common interface for all strategies and defines methods train, predict and print\_results which were from the lab.

```
from abc import ABC, abstractmethod

class ClassificationStrategy(ABC):
    """
    Abstract base class for classification strategies.
    Defines a common interface for different machine learning models to ensure consistency.
    """
    def __init__(self, model):
        """
        Initialise the strategy with a machine learning model.

        :param model: An instance of a machine learning model created by factory pattern.
        """
        self.model = model

    @abstractmethod
    def train(self, X_train, y_train):
        """
        Abstract method to train the model with the given training data.

        :param X_train: Features for training.
        :param y_train: Labels for training.
        """
        pass

    @abstractmethod
    def predict(self, X_test):
        """
        Abstract method to predict labels for the given test data.

        :param X_test: Features for prediction.
        :return: Predicted labels.
        """
        pass

    @abstractmethod
    def print_results(self, y_test, predictions):
        """
        Abstract method to print classification results.

        :param y_test: True labels.
        :param predictions: Predicted labels.
        """
        pass
```

Each strategy implements this interface and encapsulates the behavior of the specific classification algorithm, an example here for ExtraTrees

```
from sklearn.metrics import classification_report
from src.strategies.classification_strategy import ClassificationStrategy

class ExtraTreesStrategy(ClassificationStrategy):
    """
    Concrete strategy for using the Extra Trees Classifier.
    Encapsulates the training, prediction, and result evaluation for Extra Trees.
    """
    def __init__(self, model):
        super().__init__(model)

    def train(self, X_train, y_train):
        self.model.fit(X_train, y_train)

    def predict(self, X_test):
        return self.model.predict(X_test)

    def print_results(self, y_test, predictions):
        print(classification_report(y_test, predictions))
```

The models created by the factory pattern are passed into the strategies as preconfigured objects, which ensures that the strategy pattern focuses solely on the behaviour of the models. The ModelContext manages the selected strategy and invokes the methods for training, prediction and evaluation

```

from src.strategies.classification_strategy import ClassificationStrategy
from src.decorators.error_handling_decorator import ErrorHandlingDecorator
from src.decorators.logging_decorator import LoggingDecorator
from src.decorators.result_formatting_decorator import ResultFormattingDecorator
from src.decorators.timing_decorator import TimingDecorator

class ModelContext:
    """
    Context class for managing classification strategies.
    Dynamically applies decorators for additional functionality such as logging, error handling, and
    formatting.
    """
    def __init__(self, strategy: ClassificationStrategy,
                 use_logging=True,
                 use_timing=True,
                 use_error_handling=True,
                 use_formatting=True,
                 format_type='text'):
        """
        Initialise the context with a chosen strategy and optional decorators.

        :param strategy: An instance of a ClassificationStrategy.
        :param use_logging: Flag to enable logging.
        :param use_timing: Flag to enable timing.
        :param use_error_handling: Flag to enable error handling.
        :param use_formatting: Flag to enable result formatting.
        :param format_type: Desired format for results (e.g., text, JSON, table).
        """

        if use_logging:
            strategy = LoggingDecorator(strategy)
        if use_timing:
            strategy = TimingDecorator(strategy)
        if use_error_handling:
            strategy = ErrorHandlingDecorator(strategy)
        if use_formatting:
            strategy = ResultFormattingDecorator(strategy, format_type=format_type)
        self.strategy = strategy

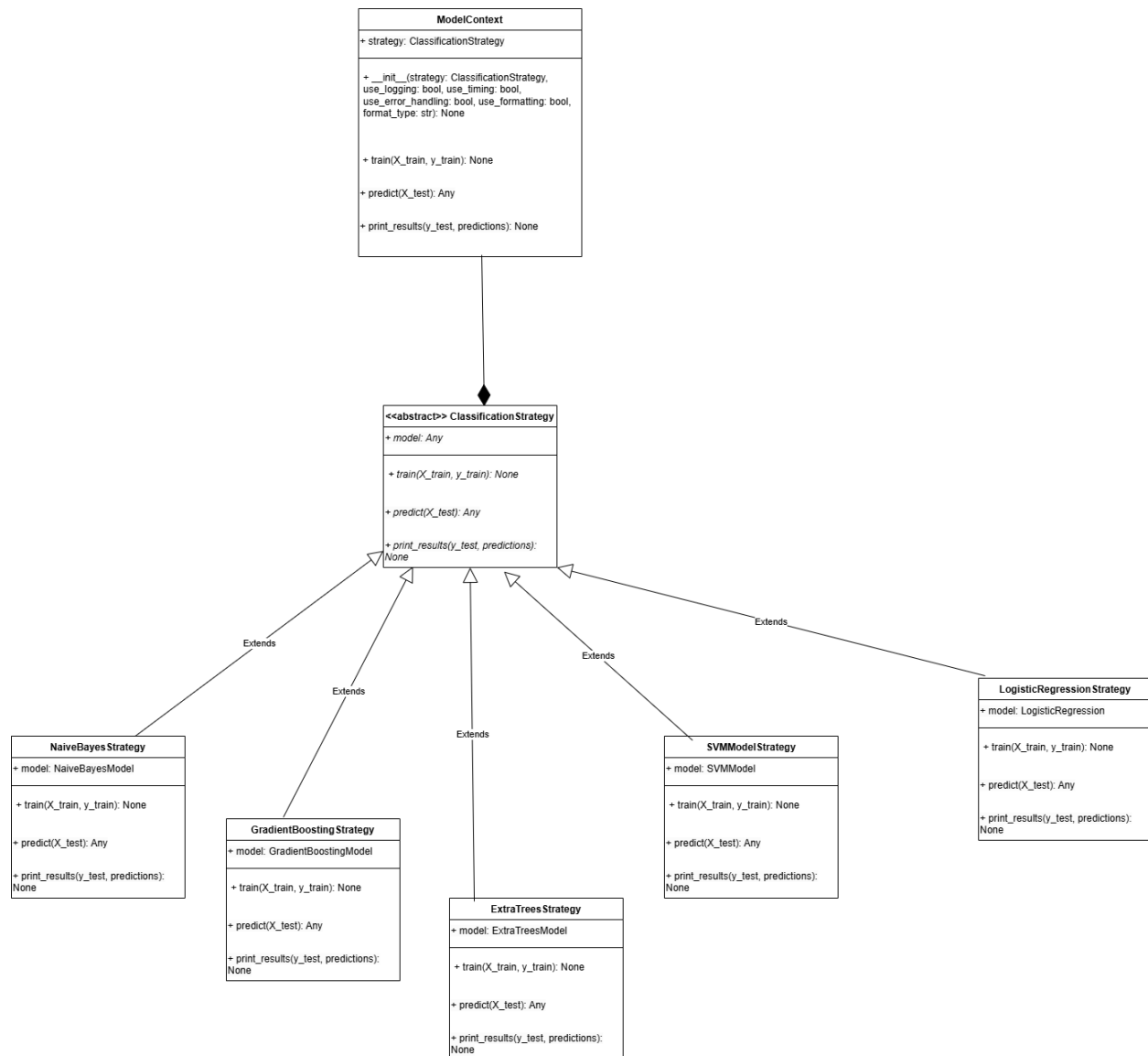
    def train(self, X_train, y_train):
        """
        Train the model using the selected strategy.
        """
        self.strategy.train(X_train, y_train)

    def predict(self, X_test):
        """
        Predict labels using the selected strategy.

        :return: Predicted labels.
        """
        return self.strategy.predict(X_test)

    def print_results(self, y_test, predictions):
        """
        Print the classification results using the selected strategy.
        """
        self.strategy.print_results(y_test, predictions)

```



## Singleton Pattern

### Purpose

The singleton pattern is implemented to ensure a single, globally accessible instance of our Configuration class (was also mentioned in lecture that this was your favorite pattern). This class centralizes some of the configuration settings. It configures hyperparameters for the machine learning models and some preprocessing options. By ensuring only one configuration class exists, it makes all the components of the system reference the same configuration which avoids inconsistencies. It also makes the settings accessible from anywhere in the application.



### *Why we chose the Singleton Pattern*

1. **Ensures single instance:** The configuration file, is loaded once and all subsequent accesses use the same instance. This avoids duplicating memory usage or redundant file reads
2. **Globally accessibility:** Any part of our application that needs to access the configuration instance can do so to retrieve the settings dynamically
3. **Flexibility and Extensibility:** By loading the settings dynamically from a JSON file, the system can be adjusted without modifying the code. For instance, you can make changes to model parameters, such as learning rate in the configuration file

### *How singleton pattern is used*

The configuration class ensures that only one instance exists by using the `__new__` method. The `__load_config` method loads the configuration settings from our JSON file at runtime. The code looks like

```

import json
import os

class Configuration:
    """
    Singleton class for managing application configuration.
    Ensures a single, globally accessible instance that reads settings from a configuration file
    (config.json).
    """
    _instance = None

    def __new__(cls):
        """
        Override the __new__ method to ensure only one instance of Configuration exists.
        If an instance doesn't exist, it creates one and initializes it.

        :return: Singleton instance of Configuration.
        """
        if cls._instance is None:
            cls._instance = super(Configuration, cls).__new__(cls)
            cls._instance._load_config()
        return cls._instance

    def _load_config(self):
        """
        Load configuration settings from a JSON file (config.json).
        If the file is not found, initializes with an empty dictionary and logs an error.

        :raises: FileNotFoundError if the configuration file is missing.
        """
        config_path = os.path.join(os.path.dirname(__file__), '.././config.json')
        try:
            with open(config_path, 'r') as f:
                self.settings = json.load(f)
        except FileNotFoundError:
            print(f"Error: Configuration file not found at {config_path}")
            self.settings = {}

    def get(self, key, default=None):
        """
        Retrieve a value from the configuration using a dot-separated key.
        Supports nested access to dictionary values.


        :param key: A string representing the configuration key (e.g., "model_params.svm.C").
        :param default: A default value to return if the key is not found.
        :return: The value associated with the key, or the default value if the key is not found.
        """
        keys = key.split('.')
        value = self.settings
        for k in keys:
            value = value.get(k, default)
            if value is default:
                break
        return value

```

And our JSON file looks like

```
{
  "model_params": {
    "naive_bayes": {
      "alpha": 1.0
    },
    "svm": {
      "C": 1.0,
      "kernel": "linear",
      "random_state": 0
    },
    "gradient_boosting": {
      "n_estimators": 100,
      "learning_rate": 0.1,
      "random_state": 0
    },
    "logistic_regression": {
      "penalty": "l2",
      "C": 1.0,
      "random_state": 0
    },
    "extra_trees": {
      "n_estimators": 1000,
      "random_state": 0,
      "class_weight": "balanced"
    }
  },
  "preprocessing": {
    "max_features": 2000,
    "stop_words": "english"
  }
}
```

Then, anywhere in the code that needs some configuration can use the 'get' method, example:



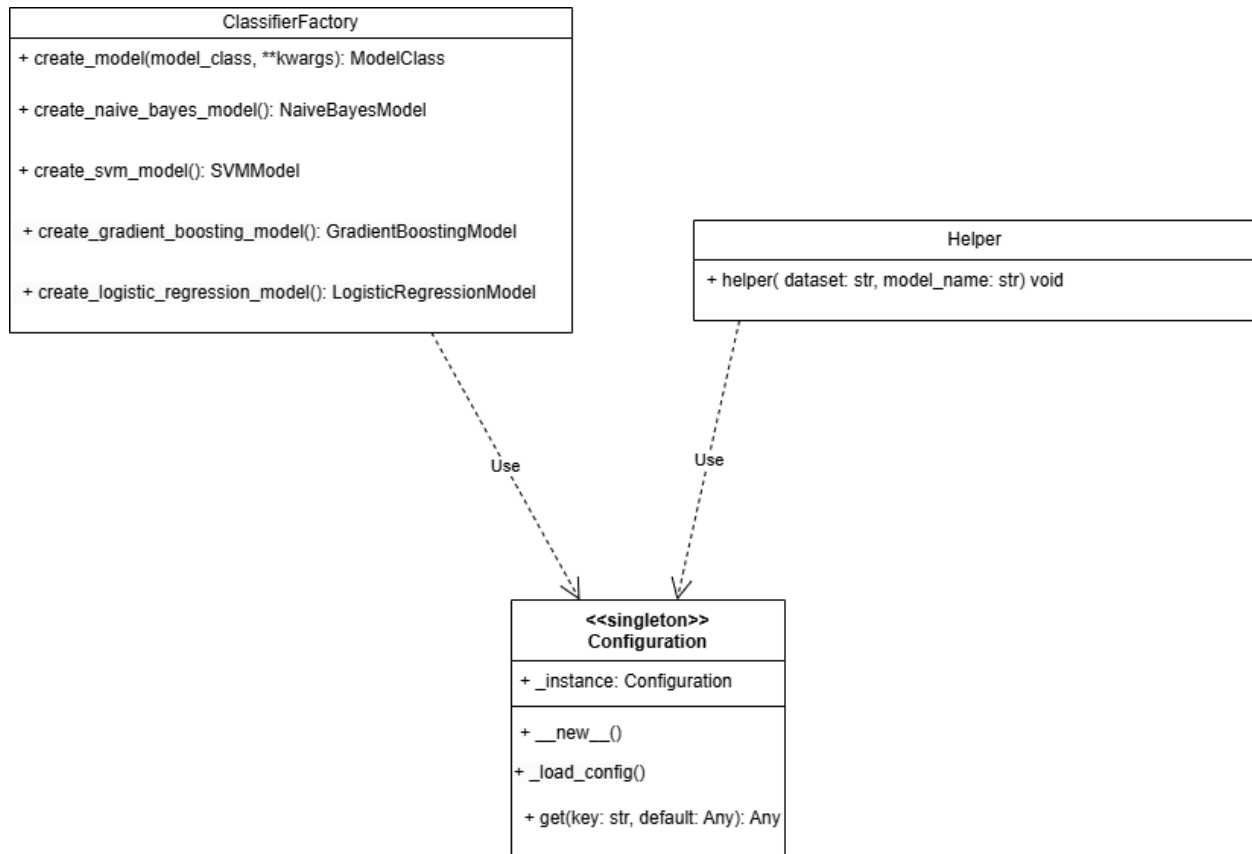
```
from sklearn.metrics import classification_report
from src.strategies.classification_strategy import ClassificationStrategy

class NaiveBayesStrategy(ClassificationStrategy):
    """
    Concrete strategy for using the Naive Bayes model.
    Encapsulates the training, prediction, and result evaluation for Naive Bayes.
    """
    def __init__(self, model):
        super().__init__(model)

    def train(self, X_train, y_train):
        self.model.fit(X_train, y_train)

    def predict(self, X_test):
        return self.model.predict(X_test)

    def print_results(self, y_test, predictions):
        print(classification_report(y_test, predictions))
```



## Observer Pattern

### Purpose

The observer pattern establishes a one-to-many dependency between objects so that when the object's state changes, all dependent objects are notified. This pattern allows for decoupling a component from its observers as the component does not need to know the details of its observers. This allows for more modular and flexible code.

### Why we choose the Observer Pattern

- **Real-Time Monitoring and Notifications:** The observer pattern allows for the system to track and log progress during the execution of the model. This is essential for a user as it gives the user an understanding of how close the model is to completing a task.
- **Decoupling of Logic:** The model should only be responsible for data preprocessing, training, and evaluation. The observer class should oversee logging. This decouples the functionality allowing the code modular and increasing maintainability.
- **Improved Debugging and Auditing:** The addition of an observer (Logger) allows for improved debugging as we are notified of what percentage of the classification is complete. This allows us to spot any issues easily if the model doesn't run until completion.

By using the Observer Pattern, we were able to create a scalable and flexible architecture for our email classification system where additional observers can easily be added in the future.

#### *How the Observer Pattern is used*

In this project, the Observer pattern is used to notify the user on what percentage of a particular model is completed when classifying a dataset.

Progress (%)	Stage	Description
10%	Dataset Loading	Dataset is loaded from the CSV file.
25%	Data Preprocessing	Data is preprocessed and split into train/test sets.
40%	Data Vectorization	Data is vectorized using TF-IDF.
50%	Model Creation	Model is created using the Factory Pattern.
65%	Model Training	Model training is completed.
80%	Model Predictions	Model makes predictions on test data.
90%	Results Saving	Results are saved to the results directory.
100%	Process Completion	Entire process is successfully completed.

```
class Observer:
    """Abstract observer class for the Observer Pattern."""
    def update(self, event_type: str, data: dict):
        raise NotImplementedError("Subclasses must implement this method.")
```

The Observer class is used to define the update function. This function should be implemented in subclasses to notify the observer of a new event.

```

class Subject:
    """Subject class to manage observers and notify them."""
    def __init__(self):
        self._observers = []

    def add_observer(self, observer: Observer):
        self._observers.append(observer)

    def remove_observer(self, observer: Observer):
        self._observers.remove(observer)

    def notify_observers(self, event_type: str, data: dict):
        for observer in self._observers:
            observer.update(event_type, data)

```

The subject class allows for observers to be added and deleted. When an event occurs the Subject calls the “notify\_observers” which triggers the update method in each observer.

```

from src.utils.observer import Observer

class Logger(Observer):
    """Logger that stores events and prints a summary at the end."""
    def __init__(self):
        self.logs = []

    def update(self, event_type: str, data: dict):
        if event_type == "start":
            self.logs.append(f"Process started with model: {data['model']} and CSV: {data['csv']}")
        elif event_type == "progress":
            self.logs.append(f"Progress: {data['progress']}%")
        elif event_type == "complete":
            self.logs.append(f"Process completed. Results: {data['results']}")
            self.print_logs()

    def print_logs(self):
        print("\n[LOGGER] Summary of Events:")
        for log in self.logs:
            print(log)

```

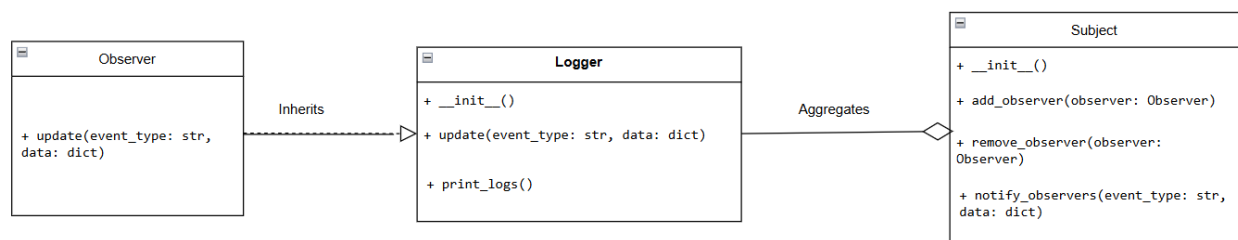
The Logger registers itself with the subject. When the subject calls the "notify\_observers" function, the loggers "update" function is triggered. There are three event types which can be called

Start: Signals the beginning of the process.

Progress: Indicates the percentage of completion.

Comple: Completion of process.

In the helper.py the subject notifies the observer of the different events which occur throughout the classification process



## Decorator Pattern

### *Purpose*

The Decorator Pattern is used to dynamically add functionality to an object at runtime without changing its underlying structure or existing code. This pattern follows the Open-Closed Principle, which allows behavior to be added without affecting the existing code. In this project, decorators we used, such as logging, timing, error handling, and result formatting, are applied dynamically to the classification strategies, allowing us to build a modular and adaptable system.

### *Why we choose the Decorator Pattern*

- **Dynamically Adding Functionality:** We wanted to add extra functionalities, like logging, result formatting, and timing, without modifying the core classification strategies. The Decorator Pattern allows us to do this by dynamically adding these additional features.
- **Staying Flexible for Future Changes:** By wrapping strategies in decorators, we avoided making changes to the core strategies. Instead, we extended their behavior, which keeps our code flexible and easy to update in the future.



- **Avoiding Too Many Classes:** Without the Decorator Pattern, each combination of features (e.g. logging & timing or timing & error handling) would require a new subclass, increasing the number of classes. With decorators, we prevent this by layering the features we want.
- **Modularity and Reusability:** Each decorator, such as LoggingDecorator or ErrorHandlingDecorator, is independent and reusable. This modularity allows us to mix and match features as needed, which makes maintenance easier.
- **Changing Features at Runtime:** Since decorators are applied at runtime, we could enable or disable functionalities (like timing or logging) dynamically based on the system's requirements or configuration.

By using the Decorator Pattern, we were able to create a scalable and flexible architecture for our email classification system where additional features can easily be added in the future.

#### *How the Decorator Pattern is used*

In this project, the Decorator Pattern is used to add functionality to the classification strategies without modifying their core implementation. Each decorator adds a specific behavior while ensuring that the original classification logic remains the same.

The classification strategies (e.g. ExtraTreesStrategy, GradientBoostingStrategy) are wrapped by several decorators. This allows for behavior layering, where multiple functionalities are added sequentially. Each decorator focuses on a single responsibility and operates independently of others. This modularity ensures that adding or removing a decorator does not affect other decorators or the core strategy.

The decorators we have in this project are:

- **Logging Decorator:** Logs key actions like training, prediction and result printing.
- **Timing Decorator:** Measures and reports the time taken for training, predicting and result printing.
- **Error Handling Decorator:** Makes sure that any errors during the training, prediction or printing results are caught and logged, maintaining the system's stability.
- **Result Formatting Decorator:** Formats the output of the classification report into different formats (Text, Json or Table) as per user preference. This decorator highlights the versatility of the Decorator Pattern, as it changes the output without interfering with the core prediction logic.

ClassifierDecorator is the base decorator class, implementing the ClassificationStrategy interface, which implements the methods, train, predict and print\_results, and wrapping around a ClassificationStrategy instance.

```
class ClassifierDecorator(ClassificationStrategy, ABC):
    """
    A base decorator class that wraps around a ClassificationStrategy object.
    This allows for dynamically adding functionalities to existing strategies.
    """

    def __init__(self, strategy: ClassificationStrategy):
        """
        Initialise the decorator with a strategy instance.

        :param strategy: An instance of a class implementing the ClassificationStrategy interface.
        """
        self._strategy = strategy

    def train(self, X_train, y_train) -> None:
        """
        This method calls the train() method of the wrapped strategy.
        """
        self._strategy.train(X_train, y_train)

    def predict(self, X_test) -> int:
        """
        This method calls the predict() method of the wrapped strategy and returns its result.

        :return: The prediction result from the wrapped strategy.
        """
        return self._strategy.predict(X_test)

    def print_results(self, y_test, predictions):
        """
        This method calls the print_results() method of the wrapped strategy and returns its
        result.

        :return: The printed result from the wrapped strategy.
        """
        return self._strategy.print_results(y_test, predictions)
```

Each decorator extends ClassifierDecorator and adds functionality to the train, predict, and print\_results methods.

For example, the Logging Decorator adds logging functionality:

- Before and after training, it logs the start and end of the process.
- During prediction, it logs the start and outputs the predictions.
- When printing results, it logs the start and end of the process.

It doesn't modify the core behavior of the strategy, instead, it enhances it with additional logging capabilities.

```
class LoggingDecorator(ClassifierDecorator):
    """
    A decorator class that adds logging functionality.
    This decorator logs during training, prediction and printing results.
    """

    def __init__(self, strategy):
        """
        Initialise the logging decorator.
        """
        super().__init__(strategy)

    def train(self, X_train, y_train) -> None:
        """
        Log the start and completion of the training process.

        :return: The training result from the wrapped strategy.
        """
        model_name = self._strategy.__class__.__name__
        print(f"[LOG] Starting training for {model_name}")
        result = super().train(X_train, y_train)
        print(f"[LOG] Training completed for {model_name}")
        return result

    def predict(self, X_test) -> int:
        """
        Log the start of the prediction process and the prediction results.

        :return: The prediction result from the wrapped strategy.
        """
        model_name = self._strategy.__class__.__name__
        print(f"[LOG] Starting prediction for {model_name}")
        predictions = super().predict(X_test)
        print(f"[LOG] Prediction result: {predictions}")
        return predictions

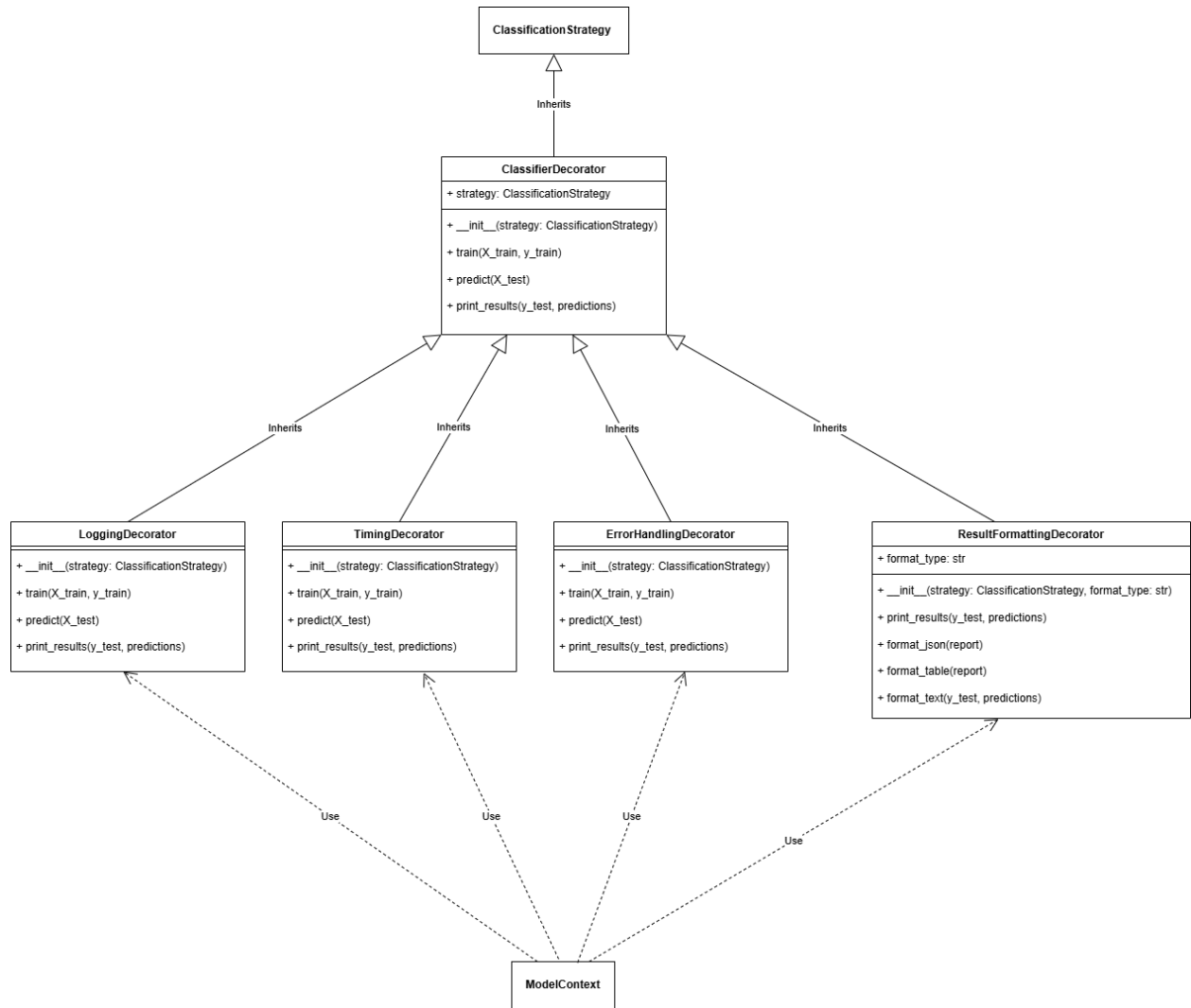
    def print_results(self, y_test, predictions):
        """
        Log the start and end of the result printing process.
        """
        print(f"\n[LOG] Printing results...")
        super().print_results(y_test, predictions)
        print(f"\n[LOG] Results printed.")
```

At runtime, when a user selects a data csv, a classification model and a preferred result format, the ModelContext dynamically applies the decorators to the selected classification strategy. The type of result format is based on the user's preference.

```
class ModelContext:
    def __init__(self, strategy: ClassificationStrategy,
                 use_logging=True,
                 use_timing=True,
                 use_error_handling=True,
                 use_formatting=True,
                 format_type='text'):

        # Apply the decorators based on the provided flags
        if use_formatting:
            strategy = ResultFormattingDecorator(strategy, format_type=format_type)
        if use_logging:
            strategy = LoggingDecorator(strategy)
        if use_timing:
            strategy = TimingDecorator(strategy)
        if use_error_handling:
            strategy = ErrorHandlingDecorator(strategy)

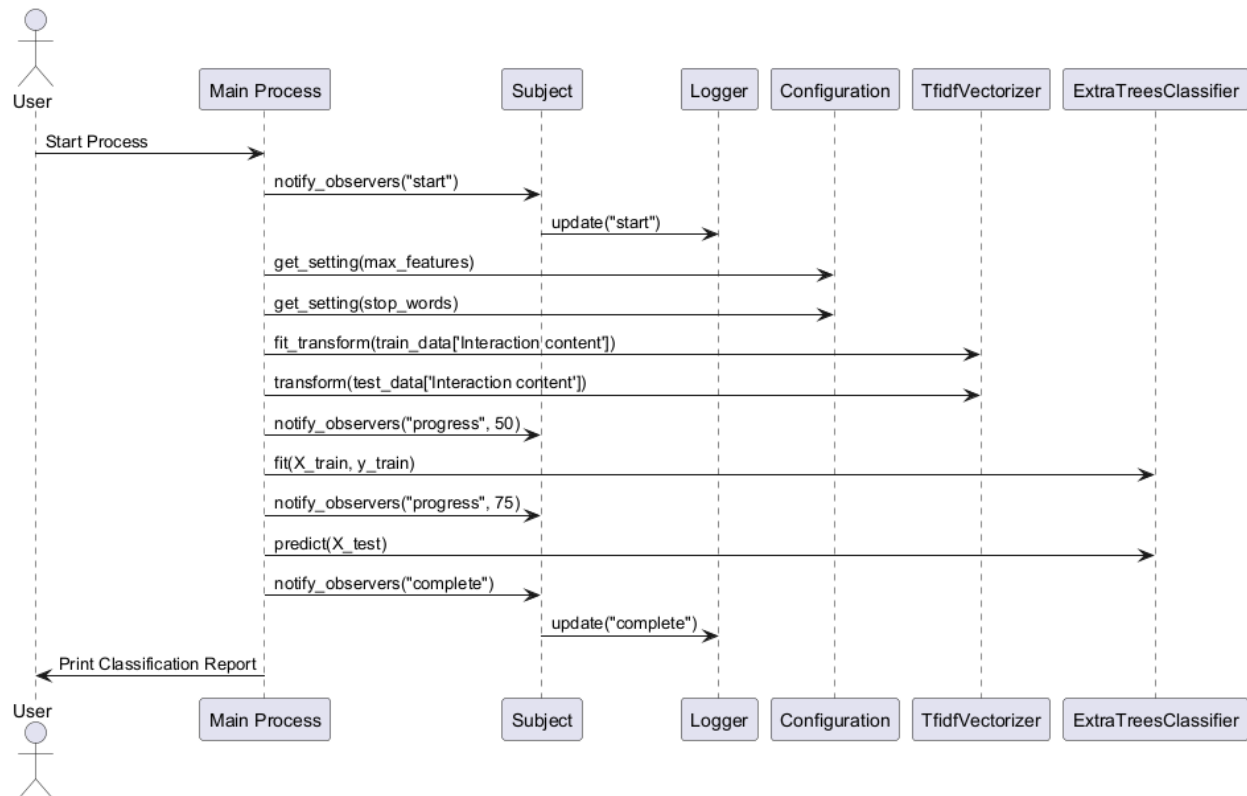
        self.strategy = strategy
```



## Email Classifiers Implemented

We used 5 models from scikit-learn library. The models are ExtraTreesClassifier, GradientBoostingClassifier, LogisticRegression, SupportVectorMachine (SVM), and NaiveBayes. These models hyperparameters can be found in the “config.json” file in the root of the project and also mentioned in our single pattern. The hyperparameters can be changed as you wish by just updating the values in the JSON files, once the model accepts this type eg making sure float input is still a float

### Sequence Diagram



#### 1. Start Process:

The user initiates the process with the CLI.

#### 2. Notify Start:

The Main Process sends a `notify_observers("start")` message to the Subject, which notifies the Logger to record the event.

**3. Configuration Retrieval:**

The Main Process retrieves configuration settings for `max_features` and `stop_words` from the Configuration.

**4. Data Preprocessing:**

The Main Process uses the `TfidfVectorizer` to

- `fit_transform` the training data into numerical form.
- transform the test data.

**5. Notify Progress:**

The Main Process sends a `notify_observers("progress", 50)` message, signaling that preprocessing is 50% complete.

**6. Model Training:**

The Main Process trains the `ExtraTreesClassifier` using the preprocessed training data (`fit(X_train, y_train)`). Decorators add extra functionality when training the classifier

**7. Notify Progress:**

The Main Process sends a `notify_observers("progress", 75)` message, signaling progress is 75% complete.

**8. Prediction:**

The `ExtraTreesClassifier` is used to predict the labels for the test data (`predict(X_test)`). Decorators add extra functionality when predicting the results.

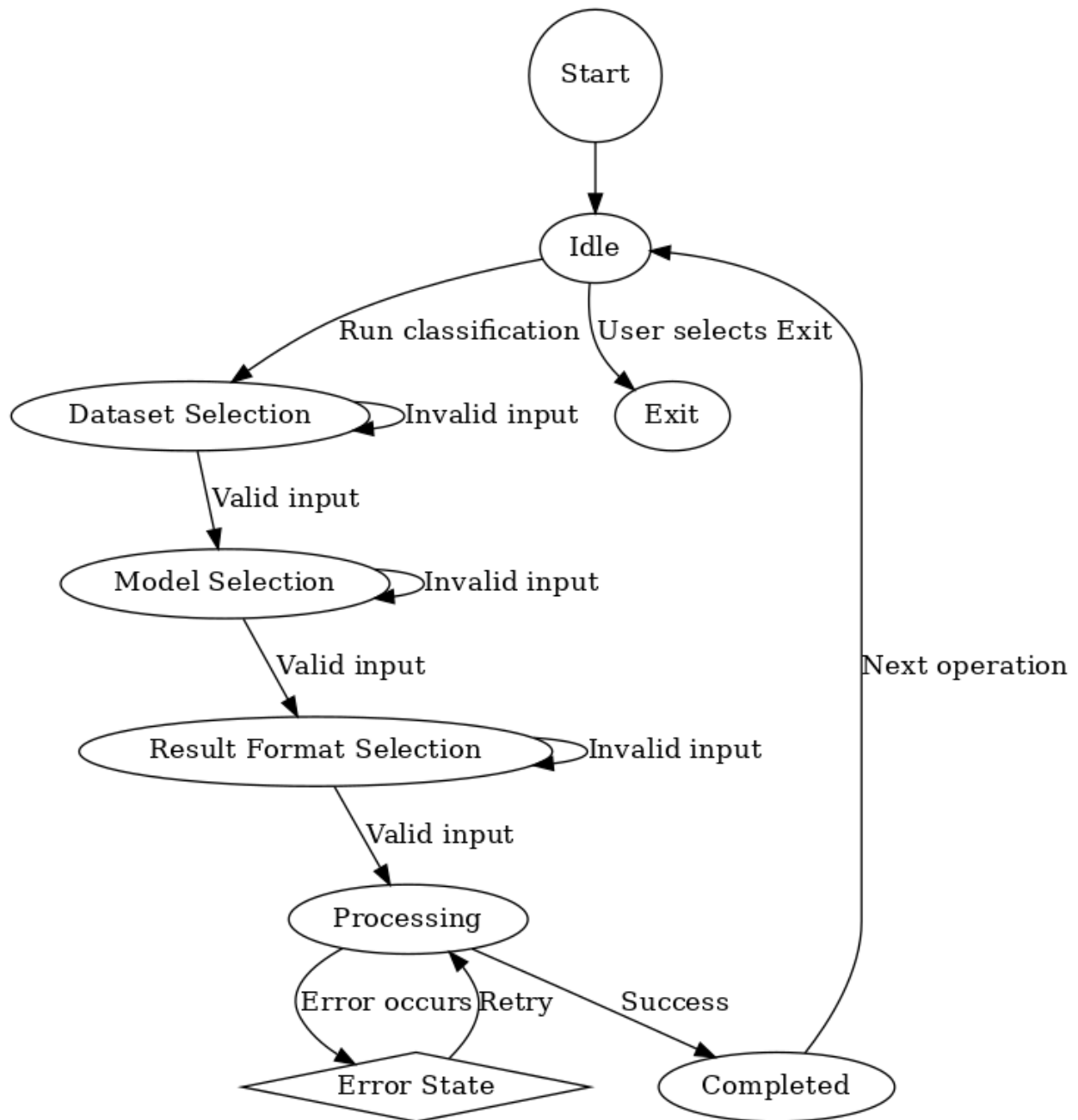
**9. Notify Completion:**

The Main Process sends a `notify_observers("complete")` message to the Subject, which informs the Logger about the completion.

**10. Print Results:**

Finally, the classification report is printed for the user. Decorators add functionality when printing results.

## State Diagram



## Use Of Agile For collaboration on our project

In the development of our email classifier, we adhered to the Agile methodology to ensure flexibility, iterative progress, and customer-centric development. We also used github for version control and our repository link can be found here: [olanhealy/Cs4125-Email-Classifier](https://github.com/olanhealy/Cs4125-Email-Classifier)



We used iterative development by dividing the project into specific areas, such as preprocessing being done first, then implementing design patterns and then improving on them and making the cli. We also heavily used the Extreme Programming technique of pair programming to help in making the design patterns and planning them out. We also continually refactored the code to enhance its readability, maintainability and performance. An example of this was when the Factory and strategy patterns were first implemented, we noticed we were essentially providing the logic for the classifiers in both the factory and strategy pattern. We refactored our code to let strategy pass in a model instead, so the factory pattern was only solely responsible for model instantiation and strategy invoked the logic.

We also used the process of code reviews to make sure our code adhered to agreed standards and help identify potential bugs. Each PR would have to get at least 2 approvals before being merged onto our 'production' branch. This review process not only improved code quality but also fostered collaboration among team members, as reviewers often suggested improvements or alternative approaches. By having multiple perspectives on the code, we were able to catch subtle issues that might have otherwise gone unnoticed.

Overall, we found that combining code reviews, pair programming, and continuous refactoring created a scalable development process. These practices, alongside our use of GitHub, helped us build a maintainable and flexible email classifier system that adheres to both best practices and our project goals.