"Astrophysics Workflow Engine - Project Report"

Prepared by: Farid Nowrouzi

Matricola No: 552343

Course Title: Computer Science

Subject: Object Oriented Programming – java

Professor : Professor Salvatore Distefano

Submission Date: 18/11/2024

University of Messina

Università degli Studi di Messina

Università degli Studi di Messina

Table of Contents

“Introduction”

Purpose Of the Project:

The Astrophysics Workflow Engine is designed to facilitate the orchestration of astrophysical data processing workflows. This project demonstrates a modular, object-oriented architecture that supports flexibility and future enhancements, such as integrating machine learning models and cloud deployment.

The Main Scope of the Project:

This project consists of a workflow engine developed in Java, using Spring Boot for microservices and JavaFX for the graphical user interface. Each workflow step, such as data processing or segmentation, is implemented as a modular microservice, allowing easy expansion and future integration of advanced features.

The key technologies Used:

The project is built using Java for object-oriented capabilities, Spring Boot for creating modular microservices, and JavaFX for the user interface. Maven is used to manage dependencies and streamline the build process.

"OOP Concepts Overview"

1. Modularity

Definition: Modularity is the design principle of dividing a system into separate, self-contained modules that can be developed, tested, and maintained independently.

Implementation in Project: Each workflow step (e.g., SegmentationStep, DetectionStep) and system feature (e.g., Orchestration, WorkflowApp) is implemented as a distinct module.

Code Example:

```
public class Workflow {

    private List<Step> steps;

    public Workflow(List<Step> steps) {

        this.steps = steps;

    }

    public void execute() throws WorkflowExecutionException {

        for (Step step : steps) {

            step.execute();

        }

    }

}
```

Explanation: The Workflow class encapsulates the logic for managing a list of steps as a cohesive unit. Each step is modular and can be added or removed from the workflow without affecting the rest of the system, demonstrating modularity.

2. Encapsulation

Definition: Encapsulation is the principle of bundling data (fields) and methods that operate on the data into a single unit (class) while restricting access to the internal details of the class.

Implementation in Project: In the project, Internal states like stepName and completed in BaseStep are encapsulated and accessed via getter and setter methods.

Code Example:

```
public abstract class BaseStep implements Step {

    protected String stepName;

    private boolean completed = false;

    public String getStepName() {

        return stepName;

    }

    public void setCompleted(boolean completed) {

        this.completed = completed;

    }

}
```

Explanation: The BaseStep class encapsulates the stepName and completed fields. Direct access to these fields is restricted, and they are managed through the getStepName and setCompleted methods, ensuring controlled manipulation of the step's internal state.

3. Inheritance

Definition: Inheritance allows a class (subclass) to inherit properties and behavior from another class (superclass), promoting code reuse and hierarchical classification.

Implementation in project: In the project, the BaseStep class serves as a parent class for specific workflow steps like DataProcessingStep, SegmentationStep, etc. These subclasses inherit shared behavior from BaseStep while implementing their own specific logic.

Code Example:

```
public class SegmentationStep extends BaseStep {
    public SegmentationStep() {
        super("Segmentation Step");
    }
    @Override
    public void execute() throws StepExecutionException {
        System.out.println("Segmenting image...");
    }
}
```

Explanation: The SegmentationStep class inherits shared behavior and properties (e.g., stepName and completed) from the BaseStep class while providing its specific implementation of the execute method.

4. Abstraction

Definition: Abstraction focuses on exposing only essential features of a system while hiding implementation details. It is often implemented using interfaces or abstract classes.

Implementation in project: In the project, The Step interface abstracts the behavior of a workflow step, which is implemented by concrete classes like DetectionStep and SegmentationStep.

Code Example:

```
public interface Step {
    void execute() throws StepExecutionException;
}
```

Explanation: The Step interface defines the contract for all workflow steps. Concrete classes implement the execute method to provide specific functionality, such as segmentation or detection, while the interface abstracts the common behavior.

5. Polymorphism

Definition: Polymorphism allows objects of different types to be treated as instances of a common parent type, enabling flexibility in method calls and behavior.

Implementation in project :The Orchestration class processes different step types (e.g., DataProcessingStep, SegmentationStep) polymorphically as Step objects.

Code Example:

```
public void start() throws OrchestrationException {
    for (Step step : steps) {
        step.execute();
    }
}
```

Explanation: In the Orchestration class, the start method iterates over a list of Step objects. Each step, regardless of its specific type, is executed through its execute method, showcasing polymorphism.

5.1- Overloading (Compile-Time Polymorphism)

Definition: *Method overloading allows multiple methods with the same name but differing parameters (type, number, or order) within the same class. The decision about which method to invoke is resolved during compile time.*

Implementation in Project: In the project, method overloading is implemented in the SegmentationStep class, where the execute method is overloaded to provide additional functionality with different parameters.

Code Example

From the SegmentationStep class:

```
@Override
public void execute() throws StepExecutionException {
    System.out.println("[SegmentationStep] Starting segmentation
process...");
    try {
        Thread.sleep(1000); // Simulating segmentation process
        System.out.println("[SegmentationStep] Segmentation completed
successfully.");
    } catch (InterruptedException e) {
        throw new StepExecutionException("Segmentation process was
interrupted.", e);
    }
}
// Overloaded method with a context parameter
public void execute(String context) throws StepExecutionException {
    if (context == null || context.isEmpty()) {
        throw new IllegalArgumentException("Context cannot be null or
empty.");
    }
    System.out.println("[SegmentationStep] Context: " + context);
    execute(); // Calls the original execute logic
}
```

Explanation:

The execute() method with no arguments runs the segmentation step without any context.

The overloaded execute(String context) method provides additional functionality by accepting a context string, which can modify or influence the segmentation behavior.

*This demonstrates **method overloading**, where the appropriate version of execute is selected at compile time based on the provided arguments***.**

5.2- Overriding (Runtime Polymorphism)

Definition: Method overriding occurs when a subclass provides a specific implementation for a method already defined in its superclass. The method to be invoked is resolved at runtime.

Implementation in Project: In your project, method overriding is heavily used in the Step subclasses (SegmentationStep, DetectionStep, etc.). Each subclass overrides the execute() method to provide specific behavior.

Code Example

From the DetectionStep class:

```
@Override

public void execute() throws StepExecutionException {

    System.out.println("Starting detection process...");

    try {

        // Simulated detection logic

        Thread.sleep(1000);

        System.out.println("Detection completed successfully.");

    } catch (InterruptedException e) {

        throw new StepExecutionException("Detection process interrupted.",
e);

    }

}
```

Explanation:

The execute() method in the DetectionStep class overrides the execute() method in the Step superclass.

This allows each step type to define its specific behavior while adhering to the common Step interface.

At runtime, the appropriate subclass implementation is invoked based on the object type, showcasing dynamic polymorphism

5.3- Subtyping (Inclusion Polymorphism)

Definition: Subtyping occurs when an object of a subclass can be used where a superclass type is expected. This allows generalized handling of different subclasses.

Implementation in Project: *The Workflow class manages a list of steps (List<Step>), but the actual objects in the list are instances of specific step subclasses (SegmentationStep, DetectionStep, etc.).*

Code Example

From the Workflow class:

```
public void execute() throws WorkflowExecutionException {
    for (Step step : steps) { // Subtyping: treating all steps
generically
        try {
            step.execute(); // Polymorphic call to the overridden
execute() method
        } catch (StepExecutionException e) {
            throw new WorkflowExecutionException("Error executing
step: " + step.getClass().getSimpleName(), e);
        }
    }
}
```

Explanation: The execute() method iterates over a list of Step objects but calls the overridden execute() method of each specific subclass (e.g., DetectionStep, SegmentationStep). This demonstrates subtyping by allowing generalized handling of all Step objects while maintaining specific behavior for each subclass.

5.4- Parametric Polymorphism (Generics)

Definition: Parametric polymorphism allows a class, method, or interface to operate on objects of different types while ensuring type safety, typically through the use of generics.

Implementation in Project: *The Workflow class uses generics to define workflows for different types of steps (Workflow<T extends Step>). This ensures that only objects of type Step or its subclasses can be added to the workflow.*

Code Example

From the Workflow class:

```
public class Workflow<T extends Step> {

    private List<T> steps = new ArrayList<>();

    public void addStep(T step) {

        steps.add(step); // Add a step of type T (must extend Step).

    }

    public List<T> getSteps() {

        return steps; // Return all steps in the workflow.

    }

}
```

Explanation:

The generic type T ensures type safety, allowing workflows to work with any subclass of Step. This provides flexibility without sacrificing correctness. For example, you could create a workflow that only accepts SegmentationStep objects or one that works with all Step subclasses.

6. Information Hiding

Definition: Information hiding restricts access to the internal details of a class, exposing only what is necessary through a controlled interface.

Implementation in project: Internal details of exceptions are hidden, exposing only meaningful messages through constructors.

Code Example:

```
public OrchestrationException(String message) {

    super(message);

}
```

Explanation: The OrchestrationException class hides low-level details while providing meaningful error messages, ensuring abstraction and simplicity for the user.

7. Composition

Definition: Composition involves building a class using references to other objects, enabling flexibility and modularity.

Implementation in project: The Workflow class demonstrates composition by maintaining a list of Step objects to manage workflow execution.

Code Example:

```
public class Workflow {
    private List<Step> steps;

    public Workflow(List<Step> steps) {
        this.steps = steps;
    }
}
```

Explanation: The Workflow class composes a list of Step objects, where each step is an independent class. This allows the workflow to include any number of steps, demonstrating composition by assembling the workflow from modular components.

8. Exception Handling

Definition: Exception handling manages runtime errors gracefully, ensuring the program remains robust and user-friendly.

Implementation in project: Custom exceptions like StepExecutionException and WorkflowExecutionException are used to handle errors during step execution and workflow processing.

Code Example:

```
public class WorkflowExecutionException extends Exception {

    public WorkflowExecutionException(String message) {

        super(message);

    }

}
```

Explanation: The WorkflowExecutionException class is thrown when a critical error occurs during workflow execution, encapsulating error details and maintaining system reliability.

9. Hierarchy

Definition: Hierarchy in OOP represents a relationship between classes, where higher-level classes manage or organize lower-level classes, creating a layered structure.

Implementation in the project: This project implements hierarchy through the relationship between the Workflow and Step classes. The Workflow class operates at a higher level, managing a collection of Step objects, while the individual Step subclasses (SegmentationStep, DetectionStep, etc.) handle the specific workflow tasks.

Code Example:

```
public class Workflow {

    private List<Step> steps;

    public Workflow(List<Step> steps) {

        this.steps = steps;

    }

    public void execute() throws WorkflowExecutionException {

        for (Step step : steps) {

            step.execute();

        }

    }

}
```

Explanation: The Workflow class manages a list of Step objects, representing a hierarchical relationship where the Workflow acts as a supervisor, orchestrating the execution of individual steps. The Step objects, implemented by subclasses like SegmentationStep and DetectionStep, represent lower-level operations in the workflow hierarchy.

This hierarchy ensures clear separation of responsibilities: the Workflow class focuses on managing the overall workflow, while the Step subclasses focus on their specific execution logic.

10. **Reusability**

Definition: Reusability is evident in the shared functionality provided by the BaseStep class. All workflow steps (DataProcessingStep, SegmentationStep, etc.) inherit common behavior and properties (e.g., stepName, completed) from this class, avoiding code duplication.

Implementation in project: Reusability is evident in the shared functionality provided by the BaseStep class. All workflow steps (DataProcessingStep, SegmentationStep, etc.) inherit common behavior and properties (e.g., stepName, completed) from this class, avoiding code duplication.

Code Example:

```
public abstract class BaseStep implements Step {

    protected String stepName;
    private boolean completed = false;

    public BaseStep(String stepName) {

        this.stepName = stepName;

    }

    public String getStatus() {

        return completed ? stepName + " is completed." : stepName + " is in
progress or not started.";

    }

}
```

Explanation: The BaseStep class encapsulates shared properties (stepName, completed) and methods (getStatus) that are inherited by all step implementations. For example, SegmentationStep and DetectionStep reuse the BaseStep logic to track completion status without redefining this functionality.

This design avoids redundancy, making the codebase easier to maintain and extend. Adding a new workflow step requires only the implementation of the specific logic, as the common behavior is inherited from BaseStep.

11. Subtyping

Definition: Subtyping is a core principle of object-oriented programming (OOP) that allows objects of a derived type (subtype) to be used in place of objects of their parent type (supertype). This principle promotes flexibility and extensibility in software design, enabling the creation of systems that are easy to maintain and expand.

Implementation in project: In the Astrophysics Workflow project, subtyping is implemented through the use of the Step interface. This interface defines a contract for all workflow steps, including the execute() method. Specific step implementations, such as SegmentationStep, DetectionStep, DataProcessingStep, and DataValidationStep, inherit from the Step interface, making them subtypes of Step.

The Workflow class uses subtyping to manage and execute steps. All steps are stored as Step objects, allowing the workflow to handle any step implementation polymorphically. The WorkflowEditor class further demonstrates subtyping by dynamically creating and adding specific step instances to the workflow based on user input, treating them as Step objects.

Code Example:

```
private void addTaskToWorkflow(String taskName, ListView<String> workflowList,
TextArea statusArea) {

    Step step; // Polymorphic reference

    switch (taskName) {

        case "Segmentation":

            step = new SegmentationStep();

            break;

        case "Detection":

            step = new DetectionStep();

            break;

        case "Data Processing":

            step = new DataProcessingStep();

            break;

        case "Data Validation":

            step = new DataValidationStep();

            break;

        default:

            throw new IllegalArgumentException("Unknown task type");

    }

    workflow.addStep(step); // Subtyping: Add subtypes as Step objects

    workflowList.getItems().add(taskName);

    statusArea.appendText("[" + getCurrentTime() + "] Added " + taskName + " to the
workflow.\n");

}
```

Explanation:

Dynamic Handling of Subtypes:

- The WorkflowEditor dynamically creates instances of specific steps (e.g., SegmentationStep) and treats them as Step objects, demonstrating subtyping.

- The Workflow class processes all steps generically as Step objects, invoking the appropriate implementation of the execute() method polymorphically.

Flexibility and Extensibility:

- The use of subtyping ensures that the system is modular and extensible. New step types can be introduced by simply implementing the Step interface without modifying the core logic of the workflow.

Practical Benefits:

- Subtyping reduces coupling between components, making the system easier to test, maintain, and extend for future requirements.

Structure of the Project Architecture and Design Section

1. Introduction to Architecture:

The architecture of the Astrophysics Workflow Engine is designed to support modularity, scalability, and flexibility, enabling the seamless execution of astrophysical data processing workflows. By leveraging object-oriented design principles and a microservices-based approach, the system ensures maintainability and future extensibility. This section provides an overview of the project's core components, the design of the workflow engine, and the orchestration of microservices.

2. Overview of Key Components

The system is composed of the following key components:

1. Workflow Engine: Responsible for managing and executing the sequence of workflow steps.

2. Microservices: Each workflow step is implemented as a standalone microservice, enabling modularity and scalability.

3. JavaFX GUI : Provides an interactive user interface for creating, managing, and monitoring workflows.

4. Orchestration Layer: Ensures proper communication and data flow between the workflow engine and microservices.

5. Exception Handling and Logging : Captures and handles errors during workflow execution while maintaining logs for debugging.

A- Workflow Engine Design

```
package com.astrophysics.workflow;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.List;

public class Workflow {
    private static final Logger logger = LoggerFactory.getLogger(Workflow.class);
    private List<Step> steps;

    public Workflow(List<Step> steps) {
        this.steps = steps;
    }

    public List<Step> getSteps() {
        return steps;
    }

    public void execute() throws WorkflowExecutionException {
        logger.info("Workflow execution started.");

        for (Step step : steps) {
            try {
                logger.info("Executing step: {}",
step.getClass().getSimpleName());
                step.execute();
                logger.info("Completed step: {}",
step.getClass().getSimpleName());
            } catch (StepExecutionException e) {
                logger.error("Error executing step: {}. Exception: {}",
step.getClass().getSimpleName(), e.getMessage());
                // Throw WorkflowExecutionException if a critical error occurs
                throw new WorkflowExecutionException("Critical error occurred
during workflow execution.", e);
            }
        }

        logger.info("Workflow execution completed successfully.");
    }
}
```

Explanation: The `Workflow` class manages the execution of a series of workflow steps, represented as a list of `Step` objects. This design demonstrates the following principles:

- **Polymorphism**: Each step is treated as a `Step` interface object, allowing different step types to be handled dynamically.

- **Encapsulation**: The steps are stored in a private `steps` field, with access controlled through the `getSteps()` method.

- **Logging and Error Handling**: Logging ensures transparency, while custom exceptions (`WorkflowExecutionException`) ensure robust error management.

The workflow engine uses subtyping to handle tasks dynamically and generically. By treating all workflow steps as Step objects, the system can process various step types polymorphically, ensuring flexibility and scalability in design.

This design ensures flexibility, scalability, and reliability in the workflow execution process.

B- Microservices Architecture

Introduction

The Astrophysics Workflow Engine employs a modular microservices architecture to facilitate the execution of astrophysical workflows. Each workflow step is implemented as an independent component (microservice), encapsulating specific tasks such as data processing, validation, segmentation, and detection. These steps are orchestrated by a central engine, ensuring a seamless and error-resilient execution flow.

Design Principles

The microservices architecture of this project adheres to the following principles:

1. **Modularity**: Each workflow step is an independent module, enabling separation of concerns and ease of maintenance.
2. **Encapsulation**: Step-specific logic is contained within individual classes, reducing dependencies between components.
3. **Scalability**: The architecture supports horizontal scaling by isolating tasks into discrete units of execution.
4. **Extensibility**: New steps can be added with minimal changes to the orchestration logic, adhering to the Open-Closed Principle.
5. **Error Resilience**: Custom exceptions and logging ensure that errors are captured and handled effectively

Core Microservices

The following microservices implement the key steps of the workflow:

1. Data Processing Step

The **DataProcessingStep** microservice simulates the preprocessing of astrophysical data. It handles potential errors gracefully and provides a foundation for integrating advanced processing algorithms in the future.

```
package com.astrophysics.workflow;

public class DataProcessingStep extends BaseStep {

    public DataProcessingStep() {

        super("Data Processing Step");

    }


    @Override

    public void execute() throws StepExecutionException {

        System.out.println("Processing data...");

        try {

            // Simulate data processing with a delay

            Thread.sleep(2000);

            // Simulate an error in data processing

            throw new StepExecutionException("Data processing error");

        } catch (InterruptedException e) {

            Thread.currentThread().interrupt(); // Restore interrupted state

        }

    }

}
```

Explanation:

- **Encapsulation:** The processing logic is contained within the execute() method.
- **Error Handling**: Errors during processing are captured using a StepExecutionException, ensuring robustness.
- **Extensibility**: Advanced processing logic can be integrated without affecting other workflow steps.

2. Data Validation Step

The DataValidationStep microservice ensures that data processed in earlier steps meets the required standards of quality and integrity.

```
package com.astrophysics.workflow;

public class DataValidationStep extends BaseStep {

    public DataValidationStep() {

        super("Data Validation Step");

    }

    @Override

    public void execute() throws StepExecutionException {

        System.out.println("Validating data...");

        // Placeholder for validation logic

    }

}
```

Explanation:

- **Modularity**: Encapsulates validation logic in a dedicated class.
- **Reusability**: Can be reused across different workflows requiring similar validation processes.
- **Extensibility**: Future enhancements can include detailed validation rules or integration with external validation tools.

3. Segmentation Step

The SegmentationStep microservice divides input data into smaller, manageable parts for detailed analysis.

```java
package com.astrophysics.workflow;

public class SegmentationStep extends BaseStep {

    public SegmentationStep() {

        super("Segmentation Step");

    }

    @Override

    public void execute() throws StepExecutionException {

        System.out.println("Segmenting image...");

        try {

            // Simulate segmentation process with a delay

            Thread.sleep(1500);

        } catch (InterruptedException e) {

            Thread.currentThread().interrupt(); // Restore interrupted state

        }

    }

}
```

Explanation:

- **Encapsulation**: Encapsulates segmentation logic within a single method.
- **Scalability**: Allows for integration of advanced segmentation algorithms, such as machine learning-based techniques.
- **Error Handling**: Ensures that interruptions are handled gracefully.

4. Detection Step

The DetectionStep microservice identifies relevant objects, such as stars or galaxies, in astrophysical data.

```
package com.astrophysics.workflow;

public class DetectionStep extends BaseStep {

    public DetectionStep() {

        super("Detection Step");

    }

    @Override

    public void execute() {

        System.out.println("Starting detection process...");

        try {

            Thread.sleep(1000);

            System.out.println("Detection in progress...");

            Thread.sleep(1000);

            System.out.println("Detection completed successfully.");

        } catch (InterruptedException e) {

            System.err.println("Detection process interrupted.");

        }

        setCompleted(true);

    }


    public String generateReport() {

        return "Detection step completed. All relevant items have been
detected and logged.";

    }

}
```

Explanation:

- **Custom Reporting**: Provides a generateReport() method for detailed detection logs.
- **Polymorphism**: Implements the execute() method dynamically within the workflow.
- **Error Handling**: Handles interruptions to maintain system stability.

Orchestration Layer

The orchestration layer coordinates the execution of all workflow steps, ensuring proper sequencing and error management. This layer dynamically invokes the `execute()` method of each step and handles any exceptions that arise during execution.

```
package com.astrophysics.workflow;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import java.util.List;

public class Orchestration {

    private static final Logger logger = LoggerFactory.getLogger(Orchestration.class);

    private List<Step> steps;

    public Orchestration(List<Step> steps) {

        this.steps = steps;

    }

    public void start() throws OrchestrationException {

        logger.info("Orchestration started.");

        for (Step step : steps) {

            try {

                logger.info("Starting execution of step: {}",
step.getClass().getSimpleName());

                step.execute();

                logger.info("Finished execution of step: {}",
step.getClass().getSimpleName());

            } catch (StepExecutionException e) {

                logger.error("Error in step: {}. Exception: {}",
step.getClass().getSimpleName(), e.getMessage());

                throw new OrchestrationException("Orchestration halted due to error in
step: "

                        + step.getClass().getSimpleName(), e);

            }

        }


        logger.info("Orchestration completed.");

    }

}
```

Explanation: The Orchestration class manages a list of Step objects, iterating over them sequentially and invoking their execute() methods. It uses SLF4J for logging, providing detailed information about each step's progress and errors. If a step fails, an OrchestrationException is thrown, halting the workflow. This approach ensures error resilience and makes it easier to debug complex workflows

Advantages of the Design

The design of the Astrophysics Workflow Engine reflects the application of key object-oriented programming (OOP) principles, emphasizing modularity, scalability, flexibility, and maintainability. This section outlines the core advantages of the system's architecture and implementation:

1. Modularity

The system is designed using a microservices-inspired architecture, where each component (or step) is implemented as a self-contained unit. By leveraging the Step interface, the workflow engine encapsulates the functionality of each task (e.g., segmentation, detection, data processing) into independent modules. This modular design offers the following benefits:

- **Separation of Concerns**: Each task focuses on a specific responsibility, reducing code complexity and improving readability.
- **Ease of Testing**: Individual steps can be tested independently, ensuring higher reliability and faster debugging.
- **Reusability**: Steps can be reused across different workflows, reducing redundancy and increasing efficiency.

2. Scalability

The system's architecture is inherently scalable due to the use of subtyping and dynamic task handling. The design supports the seamless addition of new step types by implementing the Step interface without modifying existing components. This approach enables the following:

- **Future-Proofing**: The workflow engine can easily integrate new astrophysical tasks, algorithms, or machine learning models.
- **Dynamic Growth**: As the system grows in complexity, the modular design ensures that new features do not disrupt the existing functionality.

3. Flexibility

The use of subtyping through the Step interface provides flexibility in how the workflow engine manages tasks. By treating all workflow steps polymorphically, the system can dynamically handle different step implementations without hardcoding their logic. This flexibility manifests in:

- **Dynamic Task Composition**: Tasks can be added, reordered, or removed dynamically within the workflow editor, catering to diverse user needs.
- **Customizability**: Users can define unique workflows by combining different steps in various sequences.

4. Maintainability

The adherence to OOP principles enhances the system's maintainability by reducing coupling and improving cohesion:

- **Low Coupling**: Components interact through well-defined interfaces (Step), minimizing dependencies and simplifying updates.
- **High Cohesion**: Each step is focused on a single task, making it easier to understand, modify, and extend.

5. Demonstration of Key OOP Concepts

The design showcases the practical application of several OOP principles, including:

- **Subtyping**: The Step interface allows specific implementations (e.g., SegmentationStep, DetectionStep) to be treated generically, enabling polymorphic behavior.
- **Polymorphism**: The workflow engine processes tasks dynamically by invoking the execute() method on Step objects, regardless of their specific type.
- **Encapsulation**: Each step encapsulates its logic, hiding implementation details and exposing only the necessary operations.

6. User-Friendly Interface

The workflow editor, developed using JavaFX, provides an intuitive graphical interface for composing and managing workflows. Features include:

- **Drag-and-Drop Simplicity**: Users can easily add tasks to the workflow via buttons in the editor.
- **Real-Time Feedback**: The status log and progress bar provide real-time updates on task execution, enhancing usability.

7. Extensibility

The system is designed to accommodate future advancements, such as:

- **Machine Learning Integration**: New microservices implementing machine learning models can be seamlessly added as Step subtypes.
- **Cloud Deployment**: The modular architecture supports future deployment on cloud platforms, enabling distributed processing and scalability.
- **Quantum Machine Learning**: As the field evolves, quantum models can be integrated as additional workflow steps.

8. Robust Error Handling

The system incorporates robust error handling to ensure reliability:

- **Step-Level Exception Management**: Errors occurring in individual steps (e.g., StepExecutionException) are logged and managed without crashing the entire workflow.
- **Workflow-Level Resilience**: The Workflow class gracefully handles execution failures, ensuring the system remains stable under adverse conditions.

9. Practical Application in Astrophysics

The workflow engine is tailored for processing astrophysical data, such as sky images, through tasks like segmentation and detection. Its design principles make it suitable for:

- **Data Preprocessing**: Preparing astrophysical data for further analysis or visualization.
- **Scalable Research Pipelines**: Supporting workflows of varying complexity, from basic segmentation to advanced machine learning analysis.

Conclusion of this section

The microservices architecture of the Astrophysics Workflow Engine effectively isolates responsibilities, enabling modularity and scalability. Each microservice encapsulates a specific task, while the orchestration layer ensures seamless integration and execution. This design is not only robust but also extensible, supporting the integration of advanced algorithms in the future.

C- Graphical User Interface

Introduction

The graphical user interface (GUI) of the Astrophysics Workflow Engine serves as the primary interaction point for users. Designed using JavaFX, the GUI provides an intuitive and user-friendly platform for building and managing astrophysical workflows. Its modular design allows users to seamlessly interact with microservices, enabling task creation, workflow visualization, and real-time monitoring.

This section details the GUI components, their functionality, and how they integrate with the underlying workflow engine.

GUI Design and Layout

The GUI employs a clean and efficient layout, designed using JavaFX components and FXML for better separation of concerns. The main structure is built around a **BorderPane**, which organizes the interface into distinct sections:

1. **Microservice Panel** (Left Section):
    - A vertical list of buttons representing available microservices, such as **Segmentation** and **Detection**.
    - Users can drag these buttons to the workflow pane to create new tasks.
    - Each button is styled and equipped with tooltips for better usability.
2. **Workflow Pane** (Center Section):
    - A central workspace where users design workflows by dragging and dropping microservices.
    - This pane provides real-time feedback as users arrange tasks visually, creating a dynamic and interactive experience.
3. **Properties Panel** (Right Section):
    - A placeholder for displaying details or configurations related to the selected workflow tasks.
    - This section can be expanded in the future for more advanced features.
4. **Execution Log and Workflow List**:
    - The GUI also includes an area for tracking execution logs and a list of tasks added to the workflow. These features enhance transparency and allow users to monitor the workflow creation process.

Key Functionalities

Drag-and-Drop Workflow Creation:

- The GUI enables users to design workflows by dragging microservice buttons from the microservice panel and dropping them onto the workflow pane.

- This interaction is handled by the **WorkflowController**, which dynamically creates new buttons in the workflow pane at the drop location.

Real-Time Task Management:

- Users can view the list of added tasks in the workflow list and monitor execution progress in the log area.
- Clear workflows with the "Clear Workflow" button to reset both the task list and log, providing a clean slate for new workflows.

Extensibility:

- The properties panel and workflow pane are designed for scalability. Future enhancements can include more advanced task properties, real-time workflow execution, or integration with cloud-based systems.

Code Snippets and Implementation

WorkflowApp.java:

The `WorkflowApp` class initializes the JavaFX application and loads the primary GUI layout from the FXML file.

```
package com.astrophysics.workflow;


import javafx.application.Application;

import javafx.fxml.FXMLLoader;

import javafx.scene.Parent;

import javafx.scene.Scene;

import javafx.stage.Stage;


public class WorkflowApp extends Application {


    @Override

    public void start(Stage primaryStage) throws Exception {

        FXMLLoader loader = new
FXMLLoader(getClass().getResource("/WorkflowInterface.fxml"));

        Parent root = loader.load();

        primaryStage.setTitle("Astrophysics Workflow Engine");

        primaryStage.setScene(new Scene(root, 800, 600));

        primaryStage.show();

    }


    public static void main(String[] args) {

        launch(args);

    }

}
```

This class defines the entry point for the application, ensuring that the FXML layout is loaded and displayed within a JavaFX `Stage`.

***WorkflowEditor.java*:**

The WorkflowEditor class provides a dynamic interface for managing workflow tasks.

```
package com.astrophysics.workflow;


import javafx.application.Application;

import javafx.scene.Scene;

import javafx.scene.control.*;

import javafx.scene.layout.BorderPane;

import javafx.scene.layout.VBox;

import javafx.stage.Stage;


public class WorkflowEditor extends Application {

    @Override

    public void start(Stage primaryStage) {

        primaryStage.setTitle("Astrophysics Workflow Editor");


        BorderPane root = new BorderPane();


        VBox microservicePanel = new VBox(10);

        Button segmentationButton = new Button("Segmentation");

        Button detectionButton = new Button("Detection");


        microservicePanel.getChildren().addAll(segmentationButton,
detectionButton);

        root.setLeft(microservicePanel);

        primaryStage.setScene(new Scene(root, 800, 600));

        primaryStage.show();

    }

    public static void main(String[] args) {

        launch(args);

    }

}
```

This class allows users to interact with workflow tasks using a graphical interface

WorkflowController.java:

The WorkflowController file defines the logic for handling GUI interactions, including drag-and-drop functionality.

```
package com.astrophysics.workflow.controller;

import javafx.fxml.FXML;

import javafx.scene.control.Button;

import javafx.scene.input.TransferMode;

import javafx.scene.layout.Pane;

import javafx.scene.layout.VBox;

public class WorkflowController {

    @FXML

    private VBox microserviceList;

    @FXML

    private Pane workflowPane;

    @FXML

    public void initialize() {

        workflowPane.setOnDragOver(event -> {

            if (event.getGestureSource() != workflowPane) {

                event.acceptTransferModes(TransferMode.COPY_OR_MOVE);

            }

            event.consume();

        });

        workflowPane.setOnDragDropped(event -> {

            Button draggedMicroservice = (Button) event.getGestureSource();

            Button newNode = new Button(draggedMicroservice.getText());

            newNode.setLayoutX(event.getX());

            newNode.setLayoutY(event.getY());

            workflowPane.getChildren().add(newNode);

            event.setDropCompleted(true);

            event.consume();

        });

    }

}
```

This class is responsible for enabling drag-and-drop interactions, allowing users to visually compose workflows.

Graphical User Interface Enhancements in the Astrophysics Workflow Engine(Updated Part 1):

The graphical user interface (GUI) of the Astrophysics Workflow Engine was significantly enhanced to improve user experience, interactivity, and overall functionality. These updates were designed to make the workflow editor visually appealing, intuitive, and versatile, ensuring that the tool caters effectively to the needs of users in an astrophysical context. Below is a detailed overview of the additions made to the GUI:

1. **Dynamic Task Management Panel**

- **Implementation**: The GUI now includes a dedicated task management panel with buttons for adding workflow steps dynamically. These include buttons for:
  - Segmentation
  - Detection
  - Data Processing
  - Data Validation

- **Purpose**: Users can seamlessly create workflows by selecting tasks they wish to add from the panel. Each button triggers the addition of a respective task to the workflow, displayed in a task list.

2. **Execution Progress Visualization**

- **Implementation**: A **ProgressBar** was integrated into the GUI to provide real-time feedback during workflow execution. The progress bar updates dynamically as each step in the workflow is executed.

- **Purpose**: This feature enhances user engagement by providing visual progress updates, helping users monitor task execution.

3. **Execution Log Panel**

- **Implementation**: A **TextArea** serves as an execution log, displaying step-by-step updates during workflow execution. Users can track the status of tasks, including any errors or successful completions.

- **Purpose**: The log area ensures transparency in workflow operations, allowing users to debug and understand the status of their workflows effectively.

4. **Theme Toggle Functionality**

- **Implementation**: A toggle button allows users to switch between **light** and **dark** themes dynamically. This was achieved through CSS styling and JavaFX integration.

- **Purpose**: This feature caters to user preferences and enhances usability in different environments, such as low-light settings.

5. **Export Logs Feature**

- **Implementation**: Users can export execution logs to a text file using the **Export Logs** button. This feature uses a file chooser dialog to allow users to select a save location and file name for the logs.

- **Purpose**: Exporting logs provides users with the ability to save and share execution details for further analysis or reporting.

6. **Enhanced Layout and Usability**

- **Implementation**: The GUI was reorganized into intuitive sections:
    - A microservice panel on the left for adding tasks.
    - A task list view on the right to display added tasks.
    - A log and progress section at the bottom for execution details.

- **Purpose**: This layout ensures an intuitive workflow, allowing users to navigate the application effortlessly.

7. **Interactive Workflow Execution**

- Implementation: The "Execute Workflow" button now ties into the backend logic, executing all added tasks in sequence. The integration ensures that the GUI dynamically reflects the progress and status of each task.

- Purpose: This feature bridges the gap between user interaction and system functionality, offering a seamless and interactive experience.

8. **Clear Workflow Button**

- **Implementation**: A "Clear Workflow" button was added to allow users to reset the workflow editor, clearing all tasks and logs. Confirmation dialogs ensure users do not accidentally lose their progress.

- **Purpose**: This feature provides users with the ability to start fresh without needing to restart the application.

9. **Improved Accessibility Features**

- **Implementation**: Tooltips were added to all major buttons, providing users with contextual help about each functionality.

- **Purpose**: This ensures that even new users can understand and utilize the features of the workflow editor effectively.

These enhancements collectively make the Astrophysics Workflow Engine more robust and user-friendly, catering to both novice and advanced users. The combination of dynamic task management, real-time execution feedback, and customizable appearance positions this tool as a sophisticated yet accessible solution for managing astrophysical workflows.

Conclusion of this section

The graphical user interface of the Astrophysics Workflow Engine is an essential component that empowers users to design and manage workflows intuitively. With features like drag-and-drop task creation, real-time execution logs, and modular design, the GUI provides a seamless and interactive user experience. Its extensibility ensures that future enhancements can be integrated effortlessly, making it a robust and scalable solution for astrophysical data processing.

4- Testing and Exception Handling

Introduction

Robust testing and effective exception handling are essential components of any reliable software system. In the Astrophysics Workflow Engine, comprehensive unit tests were implemented to validate the behavior of individual workflow steps, the orchestration layer, and the overall application. Exception handling was incorporated at every level to ensure that errors are managed gracefully, maintaining the stability and integrity of the workflow engine.

Testing Framework

The project employs the **JUnit 5** framework for unit testing. This widely used testing library allows for the creation of structured, maintainable, and reusable test cases. Key features of the test suite include:

- Validation of individual workflow steps under both normal and exceptional conditions.
- Integration testing of workflows and orchestration.
- Application initialization testing to ensure proper startup

Unit Testing

Unit tests were written for the following components of the workflow engine:

1. **Individual Workflow Steps**: Each workflow step was tested to ensure correct execution and error handling.

DataProcessingStepTest:

```
@Test
public void testExecute() {
    DataProcessingStep dataProcessingStep = new DataProcessingStep();
    assertThrows(StepExecutionException.class, () -> {
        dataProcessingStep.execute();
    }, "DataProcessingStep execution should throw 
StepExecutionException.");
}
```

This test verifies that the `DataProcessingStep` throws a `StepExecutionException` when an error occurs.

DataValidationStepTest

```
@Test
public void testExecute() {
    DataValidationStep dataValidationStep = new DataValidationStep();
    assertDoesNotThrow(() -> dataValidationStep.execute(),
            "DataValidationStep execution should not throw any exceptions.");
}
```

- This test ensures that the `DataValidationStep` executes without any errors under normal conditions.

- **DetectionStepTest** and **SegmentationStepTest**: Both steps were validated to confirm smooth execution without exceptions.

2- Workflow Execution: The Workflow class, which manages the execution of multiple steps, was thoroughly tested:

WorkflowTest:

```
@Test

public void testExecuteWorkflow() {

    Step processingStep = new DataProcessingStep();

    Step validationStep = new DataValidationStep();

    Workflow workflow = new Workflow(Arrays.asList(processingStep,
validationStep));

    assertDoesNotThrow(() -> workflow.execute(),

            "Workflow execution should not throw any exceptions.");

}
```

This test verifies that a sequence of steps can be executed without errors.

3- Orchestration: The orchestration layer, which handles multiple workflows, was tested for error-free execution

OrchestrationTest:

```
@Test

public void testOrchestrationExecution() {

    Step processingStep = new DataProcessingStep();

    Step validationStep = new DataValidationStep();

    Workflow workflow = new Workflow(Arrays.asList(processingStep,
validationStep));

    Orchestration orchestration = new Orchestration(workflow.getSteps());

    assertDoesNotThrow(() -> orchestration.start(),

            "Orchestration execution should not throw any exceptions.");

}
```

This test ensures that the orchestration layer executes all steps in the correct sequence without failure

4- Application Initialization: The WorkflowApp was tested to confirm that the application starts correctly:

WorkflowAppTest:

```
@Test

public void testWorkflowAppInitialization() {

    assertDoesNotThrow(() -> WorkflowApp.main(new String[]{}),

            "WorkflowApp should initialize without throwing
exceptions.");

}
```

Exception Handling

1- Exception handling is integral to the workflow engine, ensuring errors are isolated and managed effectively. The following custom exceptions were implemented:

StepExecutionException:

- Handles errors specific to individual step execution.
- Example usage in DataProcessingStep:

```
throw new StepExecutionException("Data processing error");
```

2- WorkflowExecutionException:

- Manages errors occurring during the execution of a complete workflow.
- Supports chaining to preserve the root cause of errors.

3-OrchestrationException:

- Captures errors related to the orchestration process, ensuring that system-wide issues are properly logged and communicated.

Logging

The project utilizes **SLF4J** for structured logging. Key events, such as the start and completion of workflow steps, as well as any exceptions, are logged to provide transparency and assist in debugging.

Example from Orchestration:

```
logger.info("Starting execution of step: {}",
step.getClass().getSimpleName());

logger.error("Error in step: {}. Exception: {}",
step.getClass().getSimpleName(), e.getMessage());
```

Conclusion of this section

The rigorous testing and exception-handling mechanisms implemented in the Astrophysics Workflow Engine ensure a robust and reliable system. The comprehensive test suite validates both individual components and the system as a whole, while custom exceptions and logging provide resilience and transparency. Together, these features make the workflow engine well-prepared for real-world applications and future expansions.

5- Final Summary and Reflections

Conclusion

The **Astrophysics Workflow Engine** project represents a comprehensive implementation of Object-Oriented Programming (OOP) principles in designing and building a robust, modular, and extensible software system. The primary goal of the project was to create a workflow engine capable of managing and orchestrating astrophysical data processing tasks while ensuring scalability, maintainability, and ease of use and also the use of subtyping through the Step interface ensures flexibility, modularity, and scalability, making the workflow engine adaptable to future requirements. Through diligent application of software engineering principles, the project has successfully met its objectives.

Achievements

The project achieved several milestones that reflect both technical and practical expertise:

1. **Workflow Engine Development**:
   - Designed and implemented a modular workflow engine that supports the sequential execution of astrophysical data processing tasks such as segmentation, detection, validation, and processing.
   - Incorporated polymorphism to allow for seamless integration of different task types.
2. **Graphical User Interface (GUI)**:
   - Developed an intuitive JavaFX-based GUI to enable users to visually interact with the workflow engine.
   - Features include drag-and-drop functionality, task selection, and real-time status updates, enhancing the user experience.
3. **Microservices Architecture**:
   - Followed a microservices-based design pattern, ensuring each workflow task operates as an independent, reusable module.
   - Demonstrated the principles of abstraction and modularity by isolating task-specific functionality.
4. **Testing and Exception Handling**:
   - Integrated rigorous testing using JUnit to ensure the reliability of individual components and overall workflow execution.
   - Implemented custom exception classes like WorkflowExecutionException and StepExecutionException to manage runtime errors effectively, ensuring system stability.
5. **Adherence to OOP Principles**:
   - Demonstrated key OOP concepts such as inheritance, encapsulation, polymorphism, abstraction, composition, hierarchy, and information hiding throughout the project.
   - Highlighted the importance of reusable and maintainable code through the use of the BaseStep class and interface-based design.

Challenges and Lessons Learned

Throughout the project, several challenges were encountered, which provided valuable learning opportunities:

- Integrating Components: Managing the interaction between the GUI, workflow engine, and individual tasks required careful orchestration and debugging.
- Maintaining Modularity: Ensuring each component remained independent and reusable demanded thoughtful design and adherence to OOP principles.
- Testing Complex Scenarios: Designing comprehensive JUnit tests for various edge cases required meticulous planning and understanding of the workflow's behavior.

These challenges were overcome through iterative development, rigorous testing, and the application of best practices in software engineering

Future Scope

The project provides a strong foundation for future enhancements, including:

- **Machine Learning Integration**: Incorporating machine learning models to analyze astrophysical data in real-time.
- **Cloud Deployment**: Deploying the workflow engine to the cloud for scalability and distributed processing.
- **Advanced User Interfaces**: Adding features like natural language processing (NLP) for workflow creation and real-time monitoring dashboards.
- **Quantum Computing**: Exploring quantum algorithms for astrophysical data processing to achieve performance breakthroughs.
- **Software Engineering Practices**: CI/CD pipelines, enhanced security, and logging.

These enhancements would further elevate the project's capability and align it with modern advancements in astrophysics and software engineering.

Closing Reflection

In conclusion, the **Astrophysics Workflow Engine** project successfully combines OOP principles, modular architecture, and user-centric design to address the challenges of managing astrophysical workflows. The project showcases the importance of maintainable, reusable, and scalable software design, making it a robust solution for future integration and expansion. This work serves as a testament to the potential of combining rigorous software engineering practices with innovative problem-solving to achieve impactful outcomes in scientific domains.