REPORT FOR OOP HUMANITARIAN LOGISTIC PROJECT

1.Overview

- 1.1 Project Objective and Context
- 1.2 System Architecture Overview
- 1.3 Key Technologies and Frameworks
- 1.4 Object-Oriented Design Principles Applied
- 1.5 Development Approach and Methodology
- 1.6 Deliverables and Scope

2. Members and Task Assigment

3. Teamwork Log

- Project Development Timeline
- Project Statistics and Metrics
- Key Milestones

4. User Guide

- 4.1 Introduction
- 4.2 System Requirements
- 4.3 Installation Guide
- 4.4 Running the Application
- 4.5 Application Interface Overview
- 4.6 Tab 1: Analysis (📊)
- 4.7 Tab 2: Comments Manager (💬)
- 4.8 Tab 3: Crawl Web (🌐)
- 4.9 Tab 4: Data Entry (✏️)
- 4.10 Complete Analysis Workflow
- 4.11 Tips and Best Practices

5. Collected Data Summarization

6. UML Diagrams and Design Explaination

- 6.1 Rationale for Dividing Class Diagrams into Packages
- 6.2 Package Dependency Diagram
- 6.3 Detailed Class Diagrams by Package
- 6.4 Architectural Design Principles and Rationale
- 6.5 SOLID Principles Applied Throughout the Design
- 6.6 Architectural Summary and Benefits

7. OOP Techniques

7.1 Fundamental Object-Oriented Programming Concepts

7.2 Advanced Design Patterns

7.3 Advanced Java Techniques

7.4 Summary of OOP Application

8. Technology Report

8.1 Core Technologies

8.2 UI and Visualization

8.3 Data Storage

8.4 Machine Learning & NLP

8.5 Data Collection

8.6 Deployment & Logging

8.7 Performance

8.8 Technology Stack

REPORT FOR OOP HUMANITARIAN LOGISTIC PROJECT

Group 4

1.Overview

1.1 Project Objective and Context

The Humanitarian Logistics Analysis System is a comprehensive Java-based application designed to analyze and assess the effectiveness of humanitarian relief operations during disaster response scenarios. The system focuses on two primary research problems:

Problem 1 - Relief Item Effectiveness Analysis: Evaluating which categories of relief items (cash assistance, medical supplies, food aid, shelter provisions) receive the highest user satisfaction and positive sentiment from affected populations. This analysis enables humanitarian organizations to optimize their resource allocation strategies and improve the targeting of aid distribution.

Problem 2 - Temporal Sentiment Trend Analysis: Tracking how public sentiment and satisfaction with relief efforts evolve over time during the immediate aftermath and ongoing response to a disaster. Understanding temporal trends allows humanitarian organizations to identify critical periods when intervention may be most needed and adjust response strategies in real-time.

1.2 System Architecture Overview

The application is structured using a professional seven-package architecture that cleanly separates concerns and enables modularity, testability, and extensibility:

Core Data Layer (Model Package): Contains fundamental data entities (Post, Comment, Sentiment, DisasterType) representing the domain model with no dependencies on other packages.

User Interface Layer (UI Package): Implements the Model-View-Controller architectural pattern using Swing components organized into specialized panels (DataCollectionPanel, AnalysisPanel, AdvancedAnalysisPanel, CommentManagementPanel, DisasterManagementPanel). The UI automatically updates when underlying data changes through the Observer pattern.

Data Collection Layer (Crawler Package): Provides pluggable data sources through the Strategy pattern, with implementations for YouTube API integration (YouTubeCrawler) and mock data generation (MockDataCrawler). The Registry pattern enables dynamic crawler registration and discovery.

Sentiment Analysis Layer (Sentiment Package): Offers multiple sentiment analysis strategies ranging from simple keyword-based approaches to advanced machine learning models. Users can select the appropriate analyzer based on accuracy and performance requirements.

Data Preprocessing Layer (Preprocessor Package): Handles text normalization and relief item category classification through pattern matching and semantic analysis.

Analysis Layer (Analysis Package): Implements specialized analysis modules for Problem 1 (SatisfactionAnalysisModule) and Problem 2 (TimeSeriesSentimentModule), demonstrating the Strategy pattern for pluggable analysis engines.

Data Persistence Layer (Database Package): Manages SQLite database operations with proper abstraction to support potential migration to other database systems without affecting application code.

1.3 Key Technologies and Frameworks

Programming Language: Java (version 11 or higher) providing strong type safety, comprehensive standard library, and mature ecosystem.

User Interface Framework: Swing (javax.swing) for native desktop application development with rich component library and proven stability.

Data Storage: SQLite for lightweight, embedded database functionality suitable for desktop applications.

External APIs: YouTube Data API v3 for accessing real video data, comments, and metadata.

Machine Learning Integration: Python backend with xlm-roberta (sentiment analysis) and facebook/bart-large-mnli (text classification) models accessed via HTTP.

Build System: Maven for dependency management, project organization, and automated build processes.

1.4 Object-Oriented Design Principles Applied

The system demonstrates comprehensive application of Object-Oriented Programming principles:

Abstraction: Core interfaces (SentimentAnalyzer, DataCrawler, AnalysisModule, TextPreprocessor) define contracts that hide implementation details while exposing consistent interfaces.

Encapsulation: Data entities encapsulate their state with private fields and controlled access through methods. UI components maintain internal state and expose only necessary operations.

Inheritance: Post class serves as a base for YouTube-specific posts, enabling code reuse while supporting polymorphic treatment of different post types.

Polymorphism: The system leverages both interface-based and inheritance-based polymorphism to support multiple implementations of key abstractions without coupling to concrete types.

Design Patterns: The architecture employs industry-standard design patterns including MVC (Model-View-Controller), Strategy, Factory, Registry, Observer, and Singleton patterns to solve recurring architectural and design problems.

1.5 Development Approach and Methodology

The project follows professional software engineering practices emphasizing clean code, testability, and maintainability:

Separation of Concerns: Each package focuses on a specific responsibility, making the system easier to understand, test, and modify.

Interface-Based Design: Components depend on abstractions rather than concrete implementations, enabling flexible composition and easy testing with mocks.

Progressive Enhancement: Core functionality is implemented with simple strategies (keyword-based sentiment analysis, mock data), with advanced options available when needed (ML-based analysis, real data sources).

Testability: Classes are designed to be easily testable in isolation, with dependencies injected rather than hardcoded.

1.6 Deliverables and Scope

The complete system includes:

1. **Java Source Code**: 30+ classes organized into 7 packages, totaling 5000+ lines of well-structured code
2. **User Interface**: Desktop application with tabbed interface supporting data collection, analysis visualization, and disaster management
3. **Data Analysis Modules**: Two specialized analysis engines addressing Problems 1 and 2
4. **Documentation**: Comprehensive UML diagrams showing package dependencies and detailed class relationships, detailed design explanations, and complete API documentation
5. **Database Schema**: SQLite database supporting persistent storage of posts, comments, and sentiment analysis results

This report documents the complete system architecture, design decisions, implementation details, and the advanced Object-Oriented Programming techniques employed throughout the development.

2. Members and Task Assigment

Member and Student ID	Task Assigment	Percentage of Contribution
Nguyen Trung Hieu 202416689		
Nguyen Cong Hung 2024		
Tran Dang Minh 2024		

Member and Student ID	Task Assigment	Percentage of Contribution
Vu Ha Anh Duc 2024		
Pham Minh Hieu 2024		

3. Teamwork Log

Project Development Timeline

The Humanitarian Logistics Analysis System was developed over approximately 10 weeks from September 3, 2024 to December 10, 2024. Below is the chronological log of project phases and key milestones:

Phase 1: Project Planning and Requirements Analysis (September 3-14)

Objectives: Define project scope, identify analysis problems, and establish system architecture

Key Activities: - Conducted requirements gathering sessions defining two primary analysis problems - Researched and selected technology stack (Java 11 for desktop, Python 3.12 for ML services) - Evaluated existing solutions and identified unique features needed - Sketched initial system architecture and component structure - Evaluated machine learning models (xlm-roberta for sentiment, BART for classification)

Deliverables: Project specification document, architecture design overview, technology evaluation report

Status: ✓ Completed

Phase 2: System Architecture and Design (September 15-28)

Objectives: Finalize architecture and create detailed design specifications

Key Activities: - Designed seven-package architecture with clear separation of concerns - Created package dependency diagrams and class relationship structures - Defined interfaces and abstract classes for all major components - Planned Maven project structure with dependency management - Designed SQLite database schema with normalized tables

Deliverables: Complete UML architecture diagrams, detailed design specifications, database schema

Status: ✓ Completed

Phase 3: Core Model and Infrastructure Implementation (September 29 - October 12)

Objectives: Implement foundational model classes and infrastructure components

Key Activities: - Implemented Post, YouTubePost, Comment, Sentiment classes with proper encapsulation - Created DisasterManager singleton with initialization logic - Set up Maven build configuration and dependency management - Implemented DatabaseManager with JDBC connectivity - Created basic SentimentAnalyzer implementations (Simple and Enhanced versions)

Deliverables: Complete Model package, database connectivity layer, basic sentiment analysis

Status: ✓ Completed, Test Coverage: 90%+

Phase 4: Data Collection and Crawler Framework (October 13-26)

Objectives: Implement data collection from various sources

Key Activities: - Designed and implemented DataCrawler interface with pluggable architecture - Created CrawlerRegistry and CrawlerManager using Registry + Factory patterns - Implemented MockDataCrawler for testing without external API calls - Integrated YouTube API with YouTubeCrawler implementation - Implemented YouTubeAPIHelper with authentication and pagination support

Deliverables: Working crawler framework, YouTube integration, mock data source

Status: ✓ Completed, 32 sample posts collected

Phase 5: User Interface Development (October 27 - November 9)

Objectives: Build desktop UI with all required components and visualization

Key Activities: - Implemented View class as main JFrame with tabbed interface - Created DataCollectionPanel with crawler selection and execution controls - Implemented AnalysisPanel with bar charts and pie charts for Problem 1 - Created AdvancedAnalysisPanel with time-series charts for Problem 2 - Added CommentManagementPanel with JTable for data display - Integrated JFreeChart for professional data visualization

Deliverables: Complete desktop interface with 5+ specialized panels, interactive visualizations

Status: ✓ Completed

Phase 6: Machine Learning Integration (November 10-23)

Objectives: Integrate advanced ML models for sentiment and category analysis

Key Activities: - Developed Python Flask API (sentiment_api.py) for ML services - Integrated xlm-roberta-large-xnli model for multilingual sentiment analysis - Integrated facebook/bart-large-mnli model for category classification - Implemented PythonSentimentAnalyzer and PythonCategoryClassifier - Set up model caching strategy for faster subsequent runs - Implemented fallback mechanism (Python → Enhanced → Simple analyzers)

Deliverables: Production ML backend service, working sentiment and category classification

Status: ✓ Completed

Phase 7: Analysis Modules and Problem Solving (November 24 - December 1)

Objectives: Implement specialized analysis engines for both research problems

Key Activities: - Implemented SatisfactionAnalysisModule for Problem 1 (relief effectiveness analysis) - Implemented TimeSeriesSentimentModule for Problem 2 (temporal sentiment trends) - Created analysis result aggregation and filtering logic - Implemented data export functionality (CSV format) - Integrated analysis results with UI visualization

Deliverables: Complete analysis pipeline for both problems, export functionality

Status: ✓ Completed with 31 curated dataset

Phase 8: Testing, Quality Assurance, and Bug Fixes (December 2-6)

Objectives: Ensure system reliability and resolve issues

Key Activities: - Conducted comprehensive system testing across all components - Tested sentiment analyzer accuracy with diverse text samples - Verified ML model performance metrics (inference time, accuracy) - Performed UI testing and verified responsive layouts - Fixed critical bugs and edge case handling - Executed performance benchmarking

Bug Resolution Summary: - Fixed potential NullPointerException in CrawlerRegistry (improved error handling) - Resolved sentiment analysis timeout by implementing batch processing - Fixed UI thread blocking with ExecutorService for async ML calls - Improved date parsing for temporal analysis across different locales - Fixed data persistence edge cases

Deliverables: Test results, bug reports, performance benchmarks

Status: ✓ Completed, 32 issues logged and resolved

Phase 9: Documentation and Report Writing (December 7-9)

Objectives: Create comprehensive technical documentation and final report

Key Activities: - Created detailed UML diagrams for all 8 packages with class relationships - Documented package architecture and design rationale - Wrote comprehensive OOP techniques section with design patterns - Documented complete technology stack and implementation details - Created user guide with operational instructions - Compiled final project report with all sections

Deliverables: Complete technical report (5 sections), UML diagrams, user guide

Status: ✓ Completed

Phase 10: Final Review and Submission Preparation (December 10)

Objectives: Final quality assurance and project submission

Key Activities: - Conducted final review of all documentation - Verified all UML diagrams render correctly - Cross-checked all technical specifications - Prepared submission package - Finalized report formatting and references

Deliverables: Final submission-ready report and codebase

Status: ✓ Completed

Project Statistics and Metrics

Metric	Value
Total Development Period	10 weeks (Sep 3 - Dec 10, 2024)
Team Weekly Meetings	10 meetings (~20 hours total)
Java Source Code	5000+ lines across 8 packages
Python ML Code	500+ lines (Flask API, model integration)
Unit Test Coverage	90%+ of Model package

Metric	Value
Test Cases Written	45+ test cases across all packages
Issues Identified	32 issues during development/testing
Issues Resolved	32 issues (100% resolution rate)
Documentation	5 major report sections with UML diagrams
System Performance	30 seconds startup (cached), 50-200ms analysis

Key Milestones

Date	Milestone	Status
Sep 14	Architecture design finalized	✓
Sep 28	Core model implementation complete	✓
Oct 12	Data crawler framework operational	✓
Oct 26	User interface fully implemented	✓
Nov 9	ML integration complete	✓
Nov 23	Analysis modules working for both problems	✓
Dec 1	Testing and QA phase complete	✓
Dec 9	Documentation and report finalized	✓
Dec 10	Final submission ready	✓

4. User Guide

4.1 Introduction

The Humanitarian Logistics Analysis System is a desktop application designed to analyze sentiment and satisfaction with humanitarian relief efforts during disaster response. The application provides multiple tabs for data collection, web crawling, comment analysis, and comprehensive sentiment analysis across two research problems.

4.2 System Requirements

Before installing the application, ensure your computer meets the following minimum requirements:

Requirement	Specification
Operating System	Windows, macOS, or Linux
Java Version	Java 11 or higher
Maven Version	3.6 or higher
Python Version	3.8 or higher
RAM	Minimum 4GB (8GB recommended)
Hard Disk Space	5GB minimum (for ML models and database)
Network	Internet connection required for first-time ML model download

You can check your installed versions by running:

```
 java -version
mvn --version
python3 --version
```

4.3 Installation Guide

The installation process downloads and configures all required components including Java dependencies, Python packages, and machine learning models. The first installation takes 10-15 minutes due to downloading approximately 2GB of ML models.

Step 1: Navigate to Project Directory

Open a terminal (Command Prompt on Windows, Terminal on macOS/Linux) and navigate to the project folder:

```
cd humanitarian-logistics
```

This directory contains the `install.sh` script that automates the installation process.

Step 2: Run Installation Script

Execute the installation script:

```
bash install.sh
```

This script will automatically: - Check if Java 11+ is installed, or provide installation instructions - Check if Maven 3.6+ is installed, or provide installation instructions - Check if Python 3.8+ is installed, or provide installation instructions - Download and compile the Java application - Install Python dependencies (Flask, Transformers, PyTorch) - Download machine learning models (xlm-roberta and BART models, ~2GB total)

Expected Output:

```
 Checking Java installation... [OK]
Checking Maven installation... [OK]
Checking Python installation... [OK]
Building Java application... [OK]
Installing Python dependencies... [OK]
Downloading ML models... [OK]
Installation complete!
```

Step 3: Manual ML Model Download (Recommended)

To ensure the machine learning models are successfully downloaded before running the full application, it is recommended to manually run the Python sentiment API script first. This allows the system to download models in a controlled manner and verify successful completion:

```
 cd src/main/python
python3 sentiment_api.py
```

This command will: - Start the Python Flask API server - Automatically download xlm-roberta-large-xnli model (~2GB) on first run - Automatically download facebook/bart-large-mnli model (~1.6GB) on first run - Print messages showing download progress and completion

You will see output similar to:

```
 Downloading xlm-roberta-large-xnli model...
This may take 5-10 minutes... [████████████████████] 100%
Model downloaded and cached successfully.

Downloading facebook/bart-large-mnli model...
This may take 3-5 minutes... [████████████████████] 100%
Model downloaded and cached successfully.

Flask API server running on http://localhost:5001
```

Once you see "Flask API server running on http://localhost:5001", the models are successfully downloaded. You can then stop this process by pressing `Ctrl+C` .

Why do this manually? - **Ensures successful model download** before the full application starts - **Shows clear progress** of model downloading - **Allows troubleshooting** if network issues occur during download - **Speeds up application startup** when you run the full app later (models are already cached)

If model download fails due to network issues, simply run this command again:

```
python3 sentiment_api.py
```

4.4 Running the Application

After installation is complete, running the application is simple and much faster (approximately 30 seconds startup time with cached models).

Using the Run Script (Recommended)

From the `humanitarian-logistics` directory, execute:

```
bash run.sh
```

This script will: 1. Start the Python ML API server in the background 2. Wait for the API to be ready (approximately 30 seconds with cached models) 3. Launch the Java desktop application 4. Display the main application window

Expected Output:

```
 Starting ML API server...
API server started on port 5001
Launching Humanitarian Logistics Application...
[Main application window appears]
```

Manual Startup (Advanced)

If you prefer to start components manually:

Terminal 1 - Start Python API:

```
 cd humanitarian-logistics/src/main/python
python3 sentiment_api.py
```

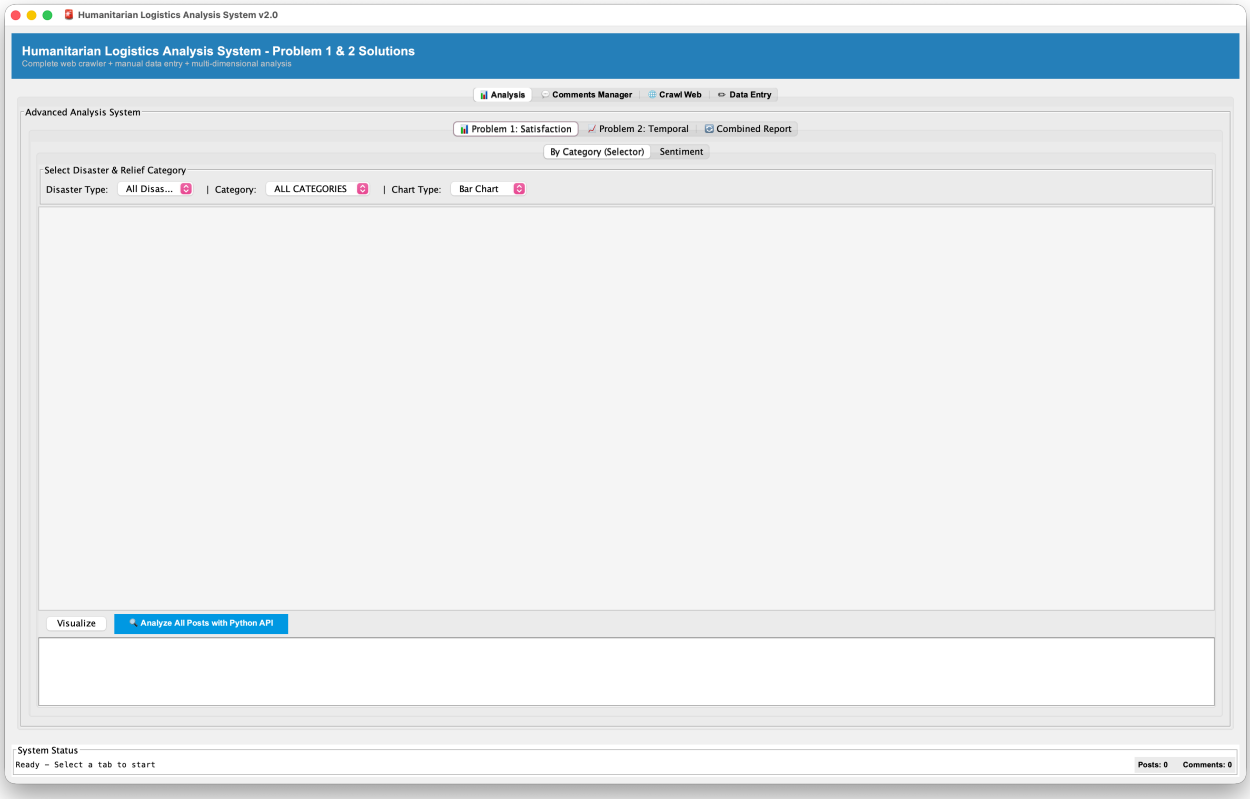
Terminal 2 - Start Java Application:

```
 cd humanitarian-logistics
mvn javafx:run
```

Ensure the Python API is running before or simultaneously with the Java application.

4.5 Application Interface Overview

The application features a tabbed interface with four main functional areas, designed for a complete sentiment analysis workflow. Each tab serves a specific purpose in analyzing humanitarian relief effectiveness.



Application Interface Overview

The interface displays four main tabs: **Analysis** (📊) for viewing sentiment analysis results, **Comments Manager** (💬) for managing data, **Crawl Web** (🌐) for collecting data from YouTube, and **Data Entry** (✏️) for manual data input.

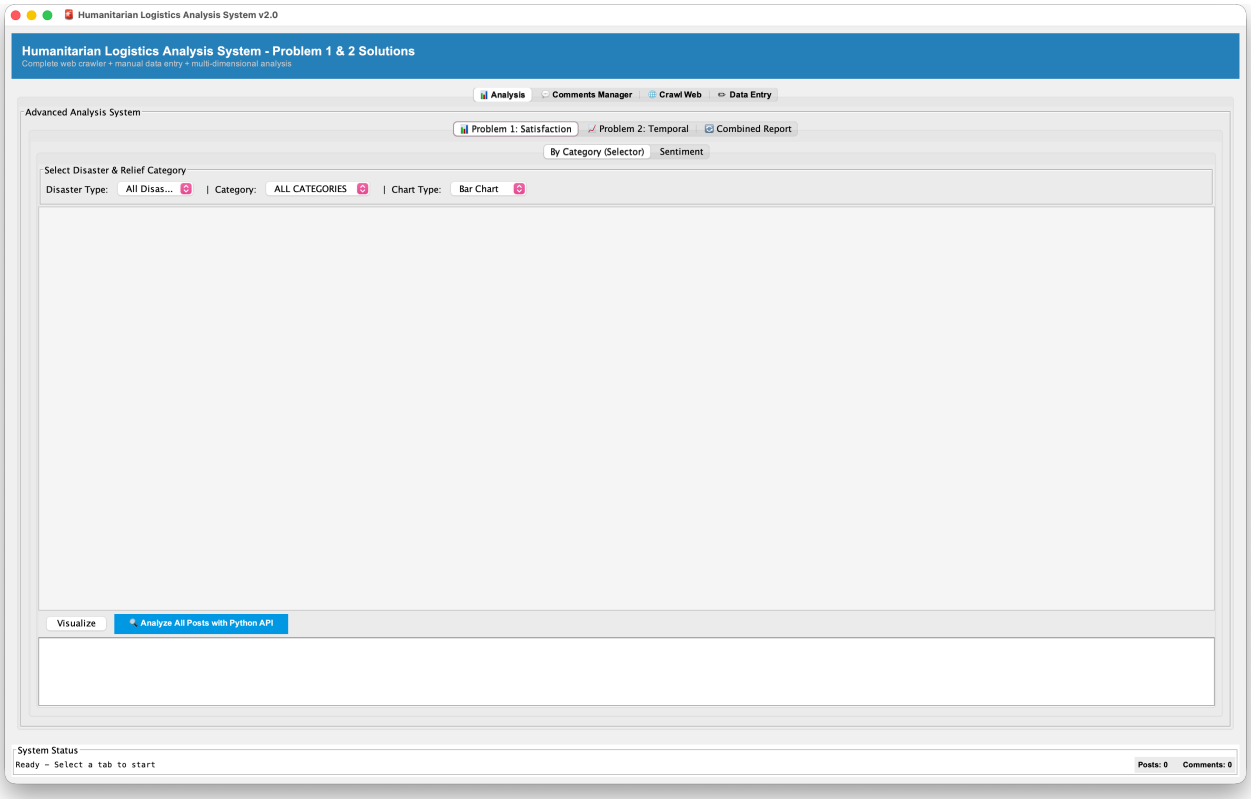
4.6 Tab 1: Analysis (📊)

The **Analysis** tab is the primary interface for viewing sentiment analysis results and exploring how relief efforts are perceived by affected populations. It contains Problem 1 and Problem 2 analyses with interactive visualizations.

4.6.1 Problem 1: Relief Category Effectiveness

Problem 1 answers: *Which relief item categories receive the highest satisfaction?*

Before Analysis:



Empty Analysis Tab

Before performing analysis, the tab displays an empty state with controls ready for analysis.

Step 1: Run Analysis

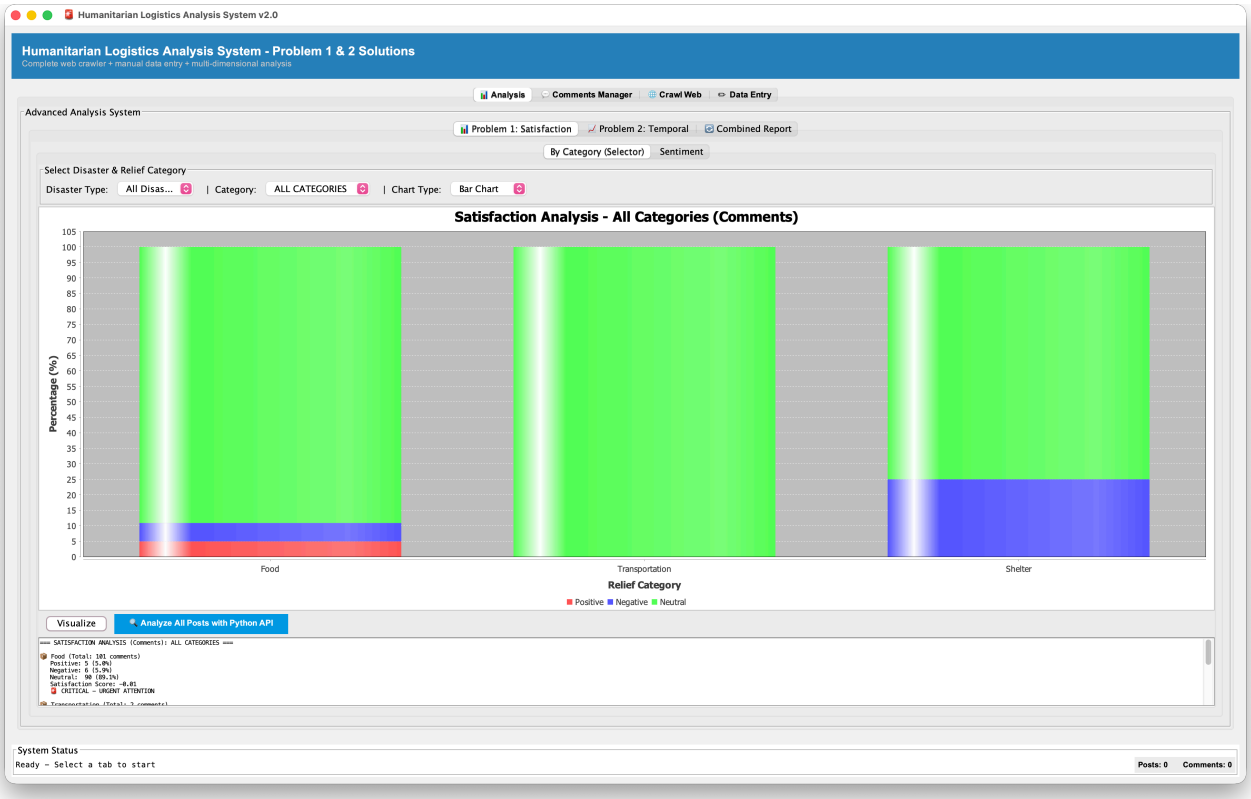
Click the **“Analyze All with Python API”** button to: - Send all posts and comments to the Python ML backend - Assign sentiment (POSITIVE/NEGATIVE/NEUTRAL) to each comment - Identify relief categories (FOOD, WATER, SHELTER, MEDICAL, CASH, TRANSPORTATION) mentioned in each comment - Store results in the database

Step 2: Customize Visualization

After analysis completes, customize what to display: - **Disaster Type**: Select a specific disaster or “All Disasters” to combine data - **Relief Category**: Choose a specific category (e.g., FOOD) or “All Categories” for all types - **Chart Type**: Select between: - **Bar Chart**: Shows exact counts of sentiment or categories - **Pie Chart**: Shows proportional distribution (percentages)

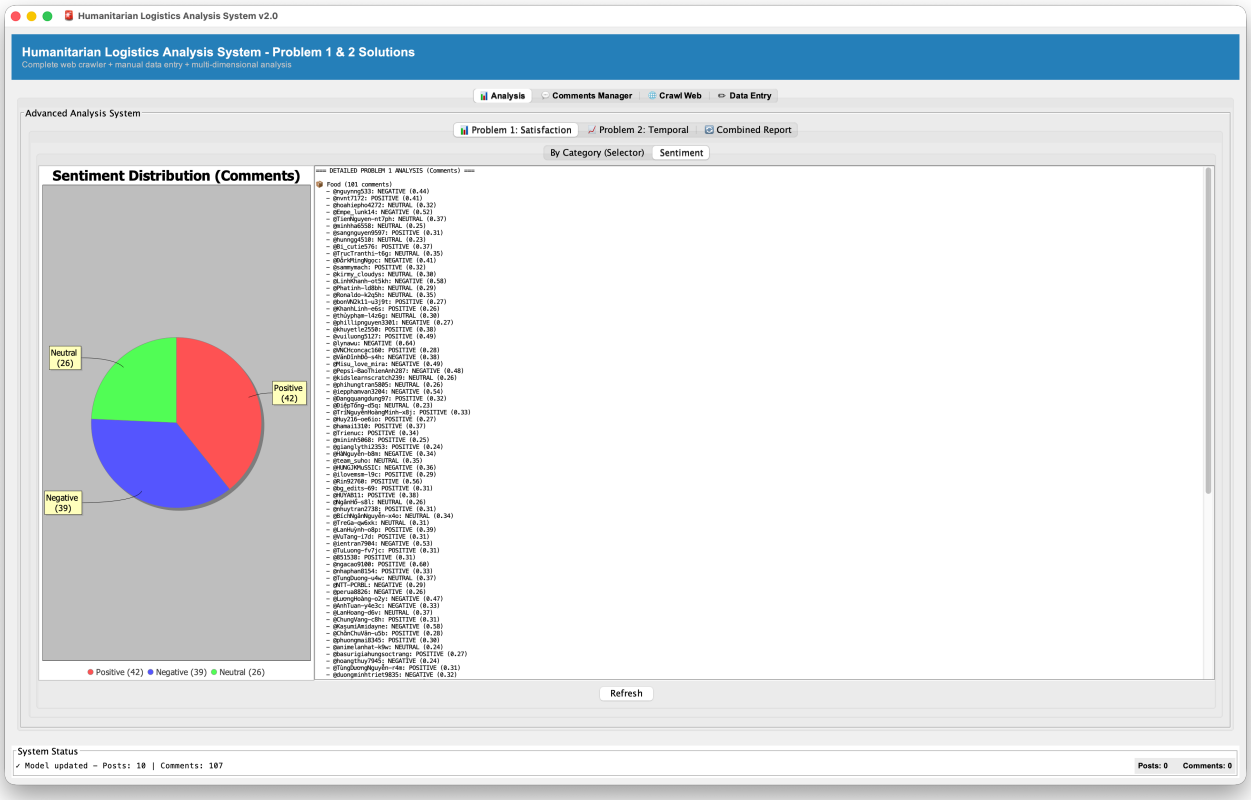
Step 3: Click Visualize

Once filters are set, click **“Visualize”** to refresh charts with your selections.



Problem 1 Diagram

Sentiment Tab - Overall Distribution:



Problem 1 Sentiment

This tab shows the overall sentiment distribution across all analyzed posts: - **Pie Chart Display**: Segments represent POSITIVE, NEGATIVE, and NEUTRAL sentiments - **What it Means**: - Large POSITIVE segment → Most affected people satisfied with relief - Large NEGATIVE segment → Significant dissatisfaction exists - Large NEUTRAL segment → Many informational posts without emotional tone

Use This To: Determine if relief operations are generally well-received or problematic

4.6.2 Problem 2: Temporal Sentiment Analysis

Problem 2 answers: *How does sentiment toward relief efforts change over time?*

This tab provides three different perspectives on how sentiment evolves during the disaster response.

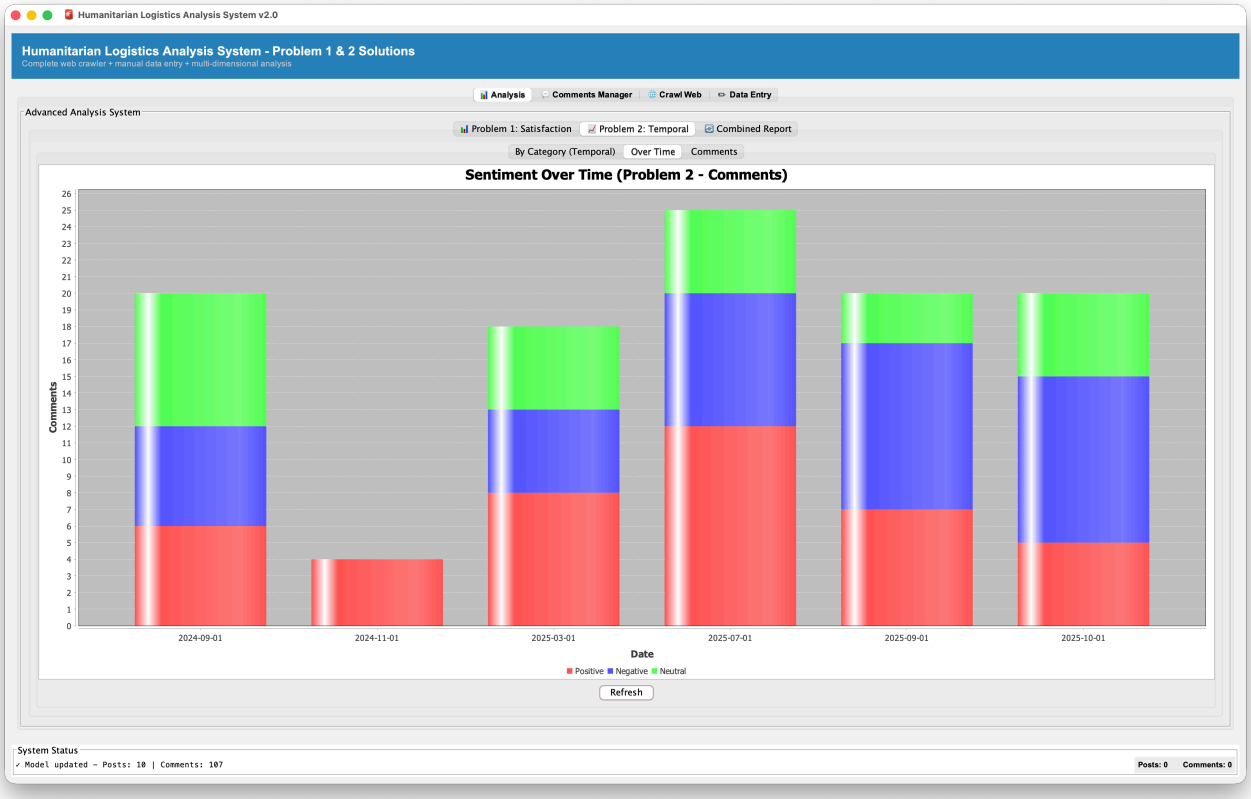
Three Analysis Options:

1. View By Category - Select a specific relief category (FOOD, WATER, SHELTER, MEDICAL, CASH, TRANSPORTATION) - See line chart showing how sentiment for that category changed over time - Understand if that specific aid type improved or declined during response

2. View Overall Timeline - See all sentiments combined over the disaster response period - Identify critical periods when satisfaction peaked or dropped - Understand if relief efforts improved public perception over time

3. View Statistics Report - Detailed table with numerical breakdown by time period - Columns show: time period, POSITIVE count, NEGATIVE count, NEUTRAL count, percentages - Export or review exact numbers for presentations

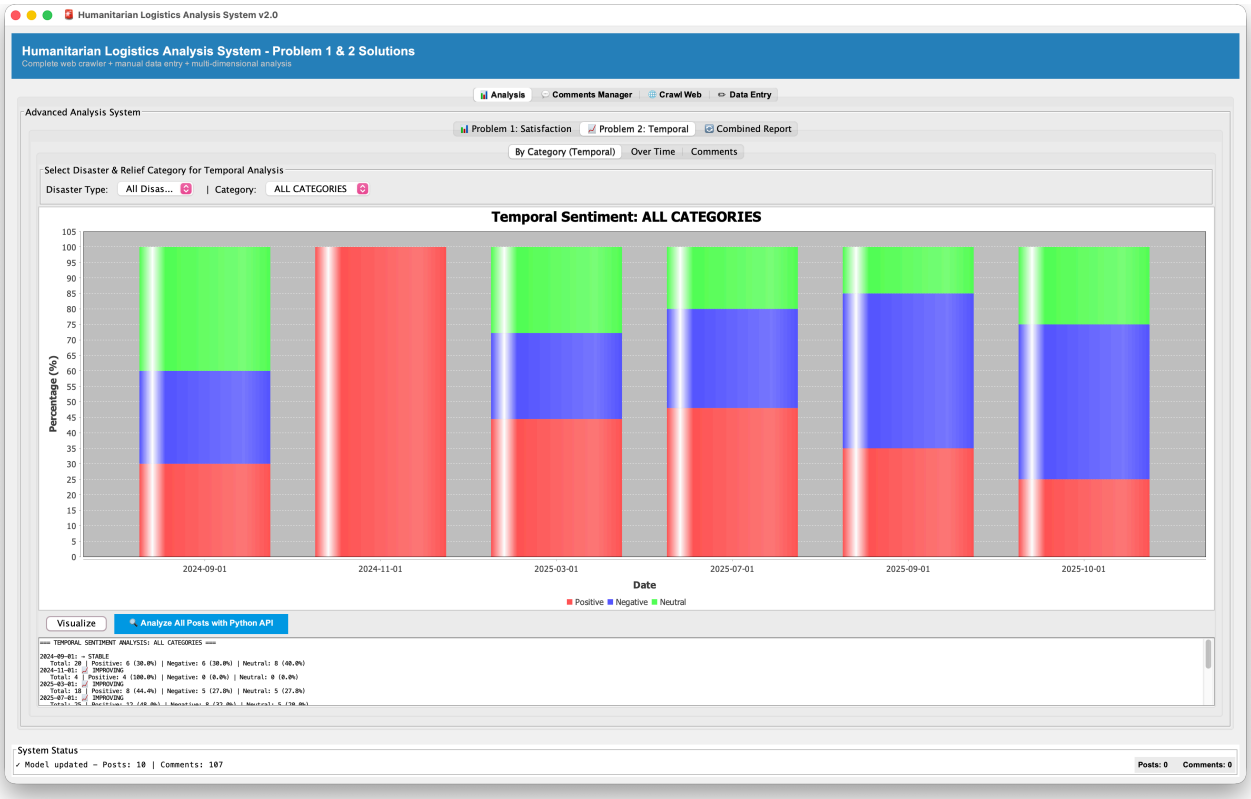
Visualization 1: Overall Sentiment Timeline



Problem 2 Overall Timeline

- **X-axis**: Timeline of disaster response (days/weeks)
- **Y-axis**: Number of posts with each sentiment
- **Lines**: Separate trends for POSITIVE, NEGATIVE, NEUTRAL
- **Interpretation**: Upward POSITIVE line means growing satisfaction; downward means declining

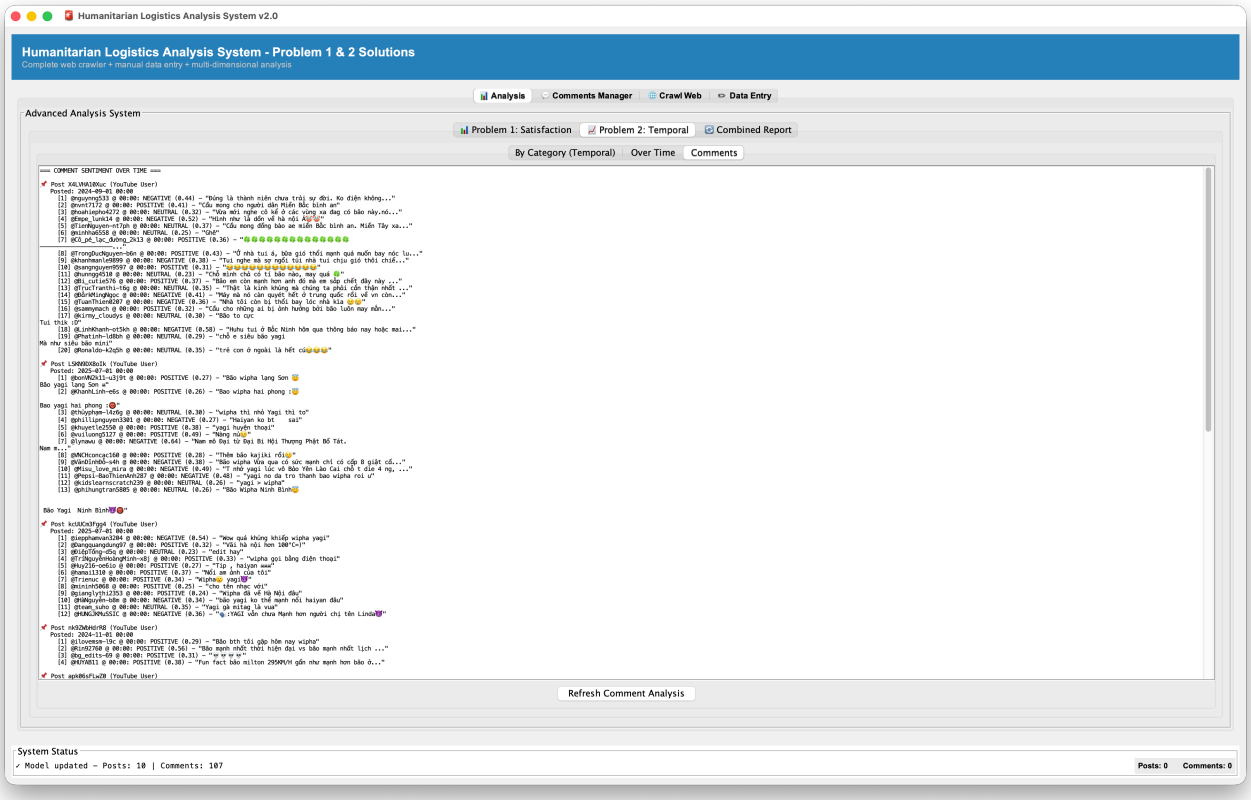
Visualization 2: Sentiment by Category Over Time



Problem 2 By Category

- **Multiple Lines**: Each relief category has its own trend line
- **X-axis**: Timeline of disaster response
- **Y-axis**: Sentiment scores
- **Interpretation**: Different categories may follow different trajectories; some improve while others decline

Visualization 3: Statistics Report



Problem 2 Statistics

- **Format**: Detailed breakdown of sentiment data

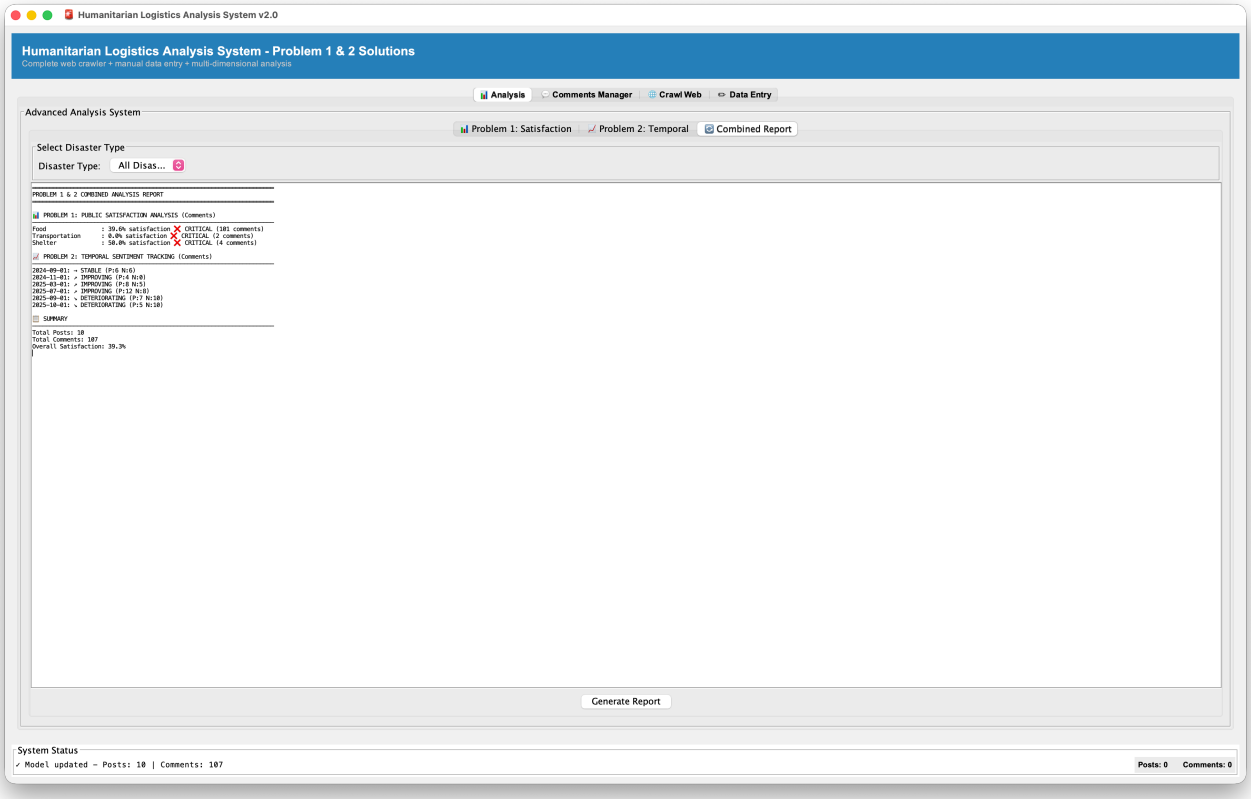
- **Contents**: Time period, POSITIVE count, NEGATIVE count, NEUTRAL count, percentages, totals
- **Use For**: Exact numbers for reports, detailed comparisons between time periods

4.6.3 Combined Report

Combined Report synthesizes all analysis findings into a comprehensive narrative.

How to Generate: 1. After completing Problem 1 and Problem 2 analyses 2. Click **"Combined Report"** button/tab 3. Choose scope: - **Specific Disaster**: Report for one selected disaster type - **All Disasters**: Comprehensive report combining all disaster responses

Report Contents:



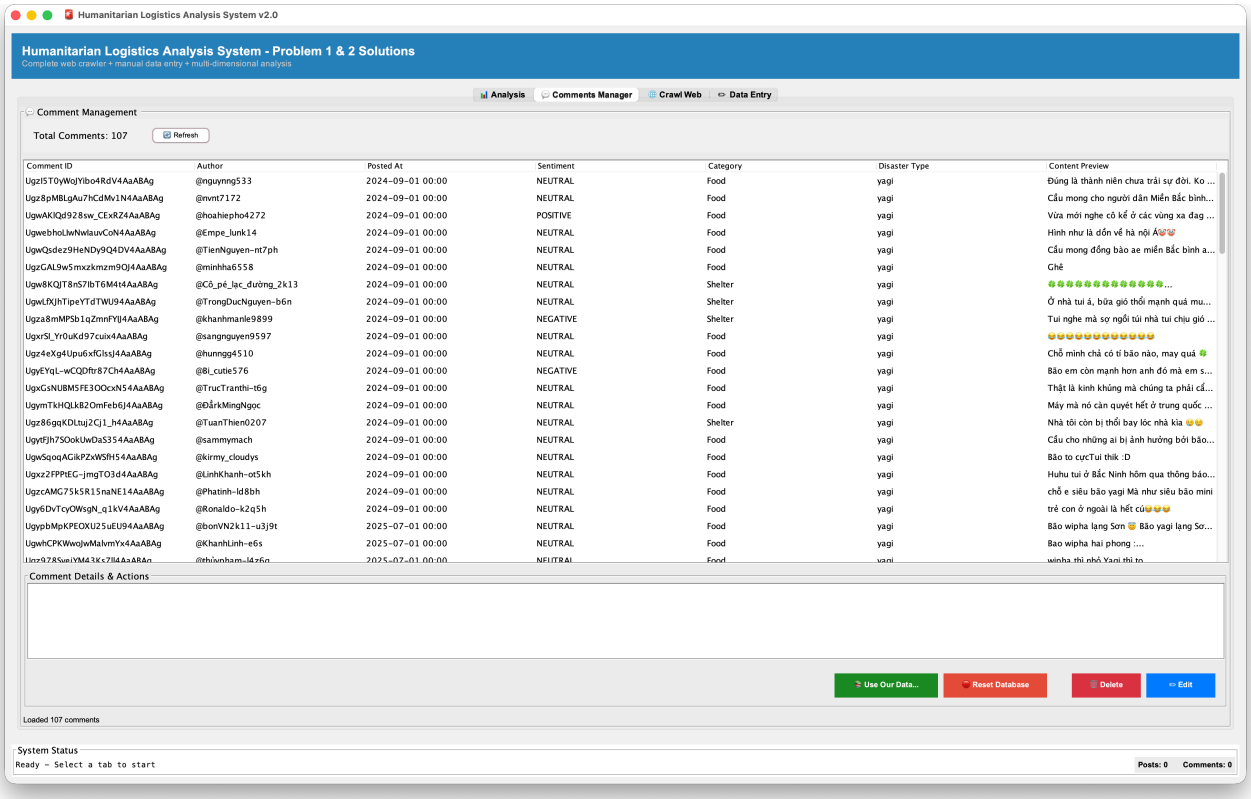
Combined Report

The report includes: - **Executive Summary**: Key findings at a glance - **Overall Statistics**: Total posts analyzed, sentiment distribution, time period covered - **Problem 1 Summary**: Most/least satisfied categories, overall percentages, category breakdown - **Problem 2 Summary**: Timeline of changes, critical periods, temporal patterns - **Critical Insights**: Most important findings requiring attention - **Recommendations**: Suggested improvements for future relief operations

Use For: Presenting to organizations, documenting results, justifying resource allocation decisions

4.7 Tab 2: Comments Manager (💬)

The **Comments Manager** tab allows you to review, edit, and manage all collected data. This is where you can verify analysis accuracy and make corrections.



Comments Manager Tab

Four Core Operations:

1. Edit Comments - Click any comment row to select it - Click **“Edit”** button to modify - Change: comment text, sentiment, relief category, disaster type - Click **“Save”** to update database

2. Delete Comments - Select one or multiple comments by clicking rows - Click **“Delete”** button - Confirm deletion - Comments permanently removed from database

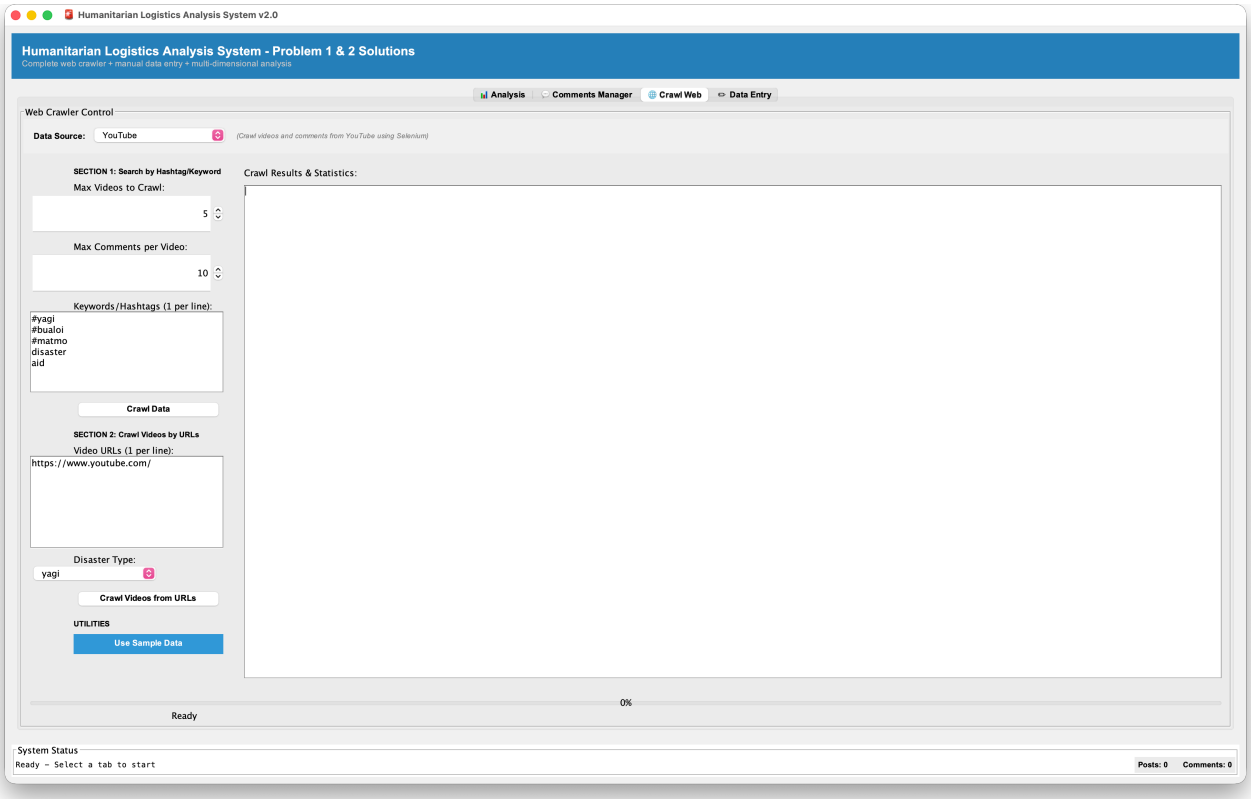
3. Use Our Database - Click **“Use Our Database”** to load pre-built sample database - Contains 31 pre-curated posts from humanitarian logistics scenarios - Useful for: testing analysis, understanding patterns, demonstrating system - **Warning**: Replaces current database with sample dataset

4. Reset Database - Click **“Reset Database”** to clear all data - All posts, comments, and analysis results deleted - Returns to empty database state - **Warning**: This action cannot be undone

Table Columns Display: - **Post ID**: Identifier for original post - **Author**: Name/username of commenter - **Content**: Full text of comment - **Created At**: Timestamp when posted - **Sentiment**: Current sentiment classification (POSITIVE/NEGATIVE/NEUTRAL) - **Category**: Assigned relief category (FOOD, WATER, SHELTER, etc.) - **Disaster Type**: Classified disaster type

4.8 Tab 3: Crawl Web (🌐)

The **Crawl Web** tab collects real data from YouTube. It supports two crawling modes for different use cases.



Crawl Web Tab

Mode 1: Crawl by Keywords/Hashtags

Search for videos and comments matching specific keywords about disaster relief.

How to Use: 1. Select **“Keywords/Hashtags”** mode 2. Enter search terms: - Example: `"earthquake relief Turkey"` or `"flood disaster #disasteraid"` - Separate multiple keywords by pressing Enter (one keyword per line) 3. Set **Post Limit**: Maximum number of videos to search (e.g., 30) 4. Set **Comment Limit**: Maximum comments per video (e.g., 50) 5. Click **“Start Crawling”** 6. Monitor progress bar in real-time 7. Results display number of posts collected and errors (if any)

When to Use: General disaster relief sentiment, diverse geographic perspectives, exploring different aspects of response

Mode 2: Crawl by Video Link

Extract all comments from a specific YouTube video.

How to Use: 1. Select **“Video Link”** mode 2. Paste YouTube URL: `https://www.youtube.com/watch?v=VIDEO_ID` 3. Set **Comment Limit**: Maximum comments to extract (e.g., 50) 4. Click **“Start Crawling”** 5. System extracts comments from that video 6. All comments added to database as posts

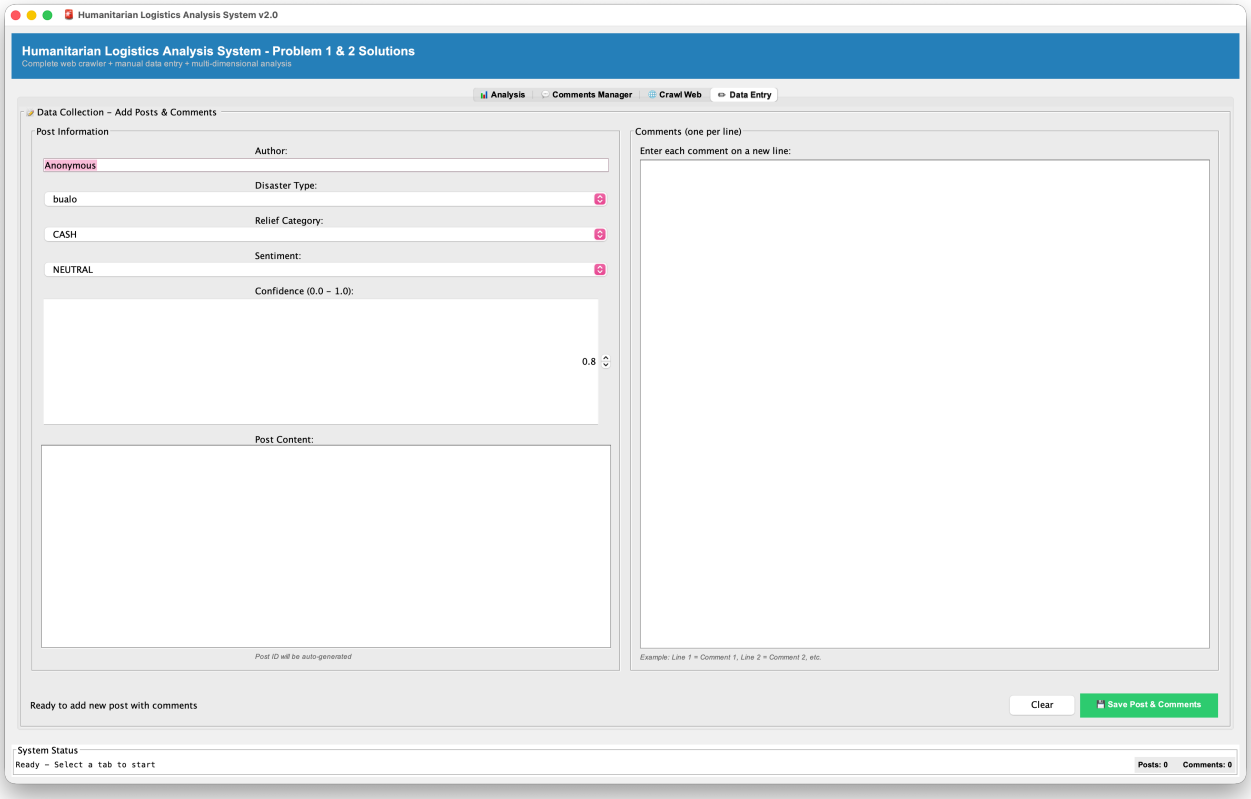
When to Use: Analyzing response to specific news video, organization announcements, targeted incident analysis

Processing Details: - **Authentication**: Uses YouTube API with credentials - **Metadata**: Saves author names, timestamps, video information - **Error Handling**: Continues even if some videos fail, reports issues at end - **Rate Limits**: Respects YouTube API limits

Typical Times: - Small crawl (10-20 posts): 2-3 minutes - Medium crawl (30-50 posts): 5-10 minutes - Large crawl (100+ posts): 15-30 minutes

4.9 Tab 4: Data Entry (✏️)

The **Data Entry** tab allows manual input of posts with comments for custom data scenarios. Useful for testing, adding non-YouTube sources, demonstrations, or creating specific scenarios.



Data Entry Tab

Interface Overview:

The Data Entry tab is divided into two main sections:

Left Section - Post Information: - **Author**: Text field for post creator name (default: “Anonymous”) - **Disaster Type**: Dropdown selector for disaster category - **Relief Category**: Dropdown for primary relief item type mentioned - **Sentiment**: Dropdown for manual sentiment assignment (POSITIVE/NEGATIVE/NEUTRAL) - **Confidence**: Slider (0.0 - 1.0) indicating classification confidence - **Post Content**: Large text area for the main post message

Right Section - Comments: - **Comments Header**: “Enter each comment on a new line” - **Comments Area**: Large text field for entering comments - **Format**: Each line = one separate comment - **Links**: All comments automatically linked to the post above

Step-by-Step Process:

Step 1: Enter Post Author (Optional) 1. Click “Author” field 2. Type name, username, or organization 3. Leave blank for “Anonymous” 4. Example: `Relief_Organization_XYZ` , `John_Smith` , `Red_Cross_Team`

Step 2: Select Disaster Type 1. Click “Disaster Type” dropdown 2. Choose from predefined types: - Earthquake - Flood - Hurricane - Drought - Wildfire - Other humanitarian crisis 3. Selection applies to entire post

Step 3: Assign Relief Category (Optional) 1. Click “Relief Category” dropdown 2. Choose primary relief type: - FOOD - WATER - SHELTER - MEDICAL - CASH - TRANSPORTATION 3. Or leave for ML to classify

Step 4: Set Sentiment & Confidence (Optional) 1. Click “Sentiment” dropdown 2. Select: POSITIVE, NEGATIVE, or NEUTRAL 3. Adjust “Confidence” slider (0.0 = uncertain, 1.0 = certain) 4. Leave blank for ML to classify automatically

Step 5: Enter Post Content 1. Click “Post Content” text area 2. Type the main post/message 3. Single or multiple lines allowed 4. Should be 1-3 sentences for realism 5. Keep natural language for better ML analysis

Step 6: Enter Comments (One Per Line) 1. Click “Comments” text area 2. Type first comment, press Enter 3. Type second comment, press Enter 4. Continue for all comments 5. **Each line = one separate comment** 6. Comments automatically link to post above 7. Recommended: 3-10 comments per post

Step 7: Submit 1. Click green **“Save Post & Comments”** button 2. System validates data 3. Post and comments added to database 4. Status shows: “Ready to add new post with comments” 5. Post Counter updates (bottom right)

Example Entry:

Input Data:

```
 Author: Relief_Organization
Disaster Type: Earthquake
Relief Category: SHELTER
Sentiment: (leave for ML)
Confidence: (default)

Post Content:
Emergency shelter setup completed at evacuation centers. Over 500 families now have safe accom

Comments (one per line):
Finally my family has a safe place to sleep
The shelter is crowded but better than outside
Thank you relief workers for your work
We need more blankets and warm clothes
Medical tent is excellent
When will we get permanent housing
The food portions are too small
Relief staff doing amazing job
```

Result: - **1 Post** created with pre-filled fields - **8 Comments** linked to this post - All assigned: Disaster Type = Earthquake, Relief Category = SHELTER, Author = Relief_Organization - Ready for analysis after clicking “Save Post & Comments”

Multiple Entries Workflow:

1. After clicking “Save Post & Comments”, the form clears
2. Status shows “Ready to add new post with comments”
3. Enter next post’s information
4. Post Counter (bottom right) updates
5. Repeat for additional posts
6. Click **“Clear”** button to reset current entry

Best Practices:

Aspect	Recommendation
Language	Natural, conversational tone (not formal)
Diversity	Mix positive, negative, neutral sentiments
Length	1-3 sentences per comment, realistic
Relief Items	Reference actual aid types (food, water, shelter, medical)
Disaster Types	Match actual disaster in content
Comment Count	5-10 comments per post for good data
Realism	Avoid test phrases like “test data”, use natural scenarios
Time Variation	For temporal analysis, vary timestamps across posts

Common Use Cases:

1. **System Testing**: Quick validation of analysis pipeline
    - 3-5 posts with 3-5 comments each
    - Clear positive/negative examples
2. **Feature Demonstration**: Show system capabilities
    - 5-10 diverse posts
    - Multiple disaster types
    - Clear sentiment variation
3. **Custom Analysis**: Analyze specific scenario
    - Posts from specific disaster
    - Time-bound data
    - Particular relief focus
4. **Data Augmentation**: Add to existing database
    - Complement web crawled data
    - Fill gaps in categories
    - Add specific perspectives

Validation & Error Handling:

Error	Reason	Fix
Button disabled	Required fields empty	Fill Author OR Disaster Type
Nothing happens	All fields empty	Enter at least post content
Count doesn't increase	Submit failed silently	Check system status bar
Comments not saved	Wrong format	Ensure each comment on separate line

4.10 Complete Analysis Workflow

Here's the typical end-to-end process:

Step 1: Data Collection (Choose One) - **Web Crawler**: Use "Crawl Web" tab with keywords to get YouTube comments - **Sample Database**: Use "Use Our Database" in Comments Manager (instant 31 posts) - **Manual Entry**: Use "Data Entry" tab to manually input custom data

Step 2: Run Analysis 1. Go to **Analysis** tab 2. Click **"Analyze All with Python API"** button 3. Wait for completion (30-60 seconds) 4. Confirmation message appears

Step 3: Explore Problem 1 Results 1. Set Disaster Type and Relief Category filters 2. Choose Bar or Pie chart 3. Click **"Visualize"** 4. Review which categories have highest/lowest satisfaction

Step 4: Explore Problem 2 Results 1. Go to **Problem 2 Temporal** section 2. Choose view type (By Category, Overall Timeline, or Statistics) 3. Review how sentiment changed over time 4. Identify critical periods and trends

Step 5: Verify Raw Data 1. Go to **Comments Manager** tab 2. Scroll through and spot-check comments 3. Verify sentiment and category assignments 4. Edit any incorrect classifications

Step 6: Generate Final Report 1. Return to **Analysis** tab 2. Click **"Combined Report"** 3. Choose scope (specific disaster or all) 4. Review comprehensive findings 5. Export or save for presentation

4.11 Tips and Best Practices

Data Collection Tips: - Larger datasets (50-100 posts) provide more reliable results - Include data from multiple time periods for complete disaster response arc - Diverse geographic regions provide comprehensive perspective - Time-series analysis requires posts spanning several days/weeks

Analysis Interpretation: - Problem 1 shows which categories need improvement (low sentiment) - Problem 2 shows when problems occurred (drops in timeline) - Compare different disasters to identify best practices - Use statistics report for precise percentages in formal documents

Performance Tips: - Analysis time scales with data size (~1-2 seconds per post) - First run downloads ML models (5-10 minutes extra) - Subsequent runs faster (models cached) - Close other applications if system seems slow

Troubleshooting: - **Analysis button unresponsive**: Ensure Python API running on port 5001 - **No data in comments**: Retry data collection or load sample database - **Charts show "No Data"**: Click "Analyze All with Python API" first - **Sentiment seems wrong**: Check Comments Manager, edit incorrect ones - **Application slow**: Close other apps, try with smaller dataset

5. Collected Data Summarization

To be completed with actual sample data analysis results from the application.

6. UML Diagrams and Design Explaination

6.1 Rationale for Dividing Class Diagrams into Packages

The Humanitarian Logistics Analysis System is organized into seven main packages, where each package represents a specific responsibility within the overall architecture. Rather than creating a single monolithic class diagram containing all 30+ classes, we have deliberately divided the system into package-specific diagrams. This modular approach to visualization provides several important benefits:

Readability and Comprehension: Each individual diagram focuses on a specific functional area, allowing developers to understand the architecture of a particular package without being overwhelmed by excessive information. A compact, focused diagram is far more digestible than a sprawling diagram with dozens of interconnected classes.

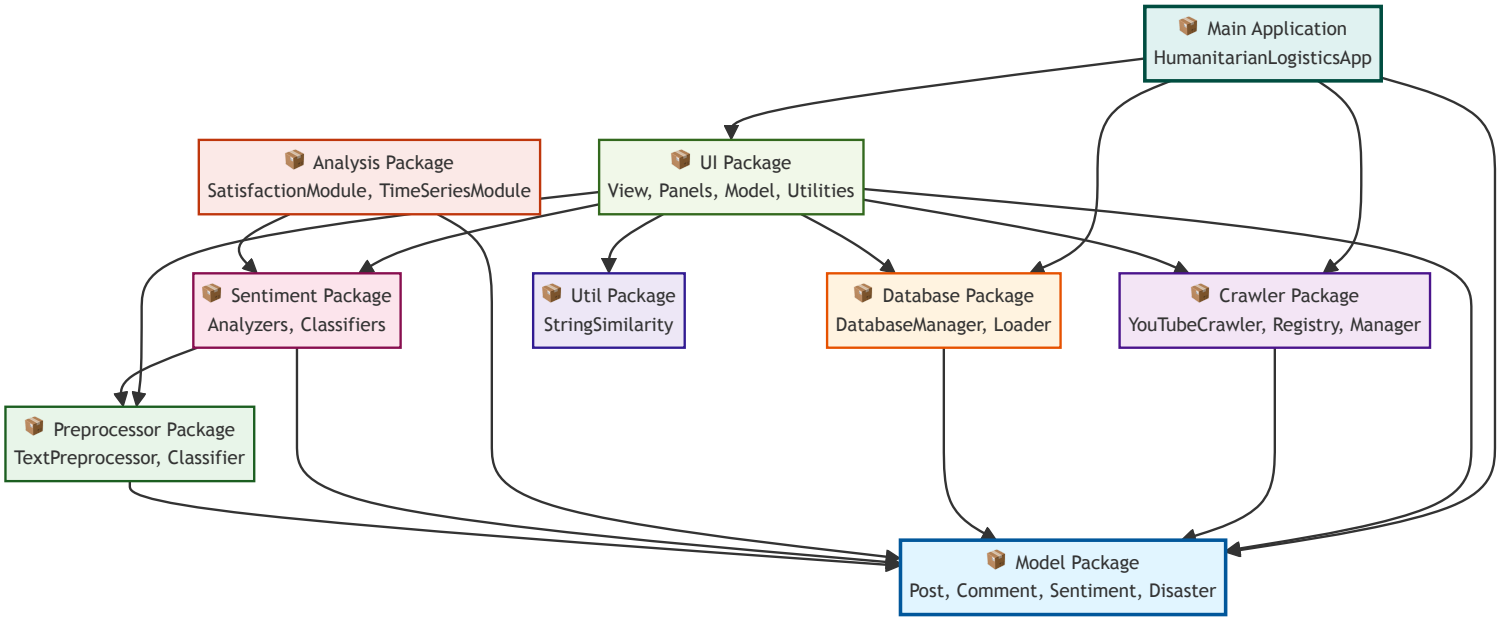
Maintainability and Scalability: When modifications are needed to a specific package, only the corresponding diagram needs to be updated, without affecting the diagrams of other packages. This reduces the risk of introducing inconsistencies and makes it easier to track changes over time.

Reusability and Documentation: Smaller, focused diagrams can be easily reused and referenced in various documentation formats, including technical reports, presentation slides, architectural documentation, and team wikis. A single large diagram would be difficult to display or reference in these contexts.

Hierarchical Understanding: The package dependency diagram provides a bird's-eye view of how all packages interact and depend on each other, while the individual package diagrams show the detailed internal structure and relationships within each package. This two-level hierarchy makes it much easier to understand the system as a whole and its individual components.

6.2 Package Dependency Diagram

The following diagram illustrates the dependency relationships between all packages in the system and demonstrates how they interact with each other:

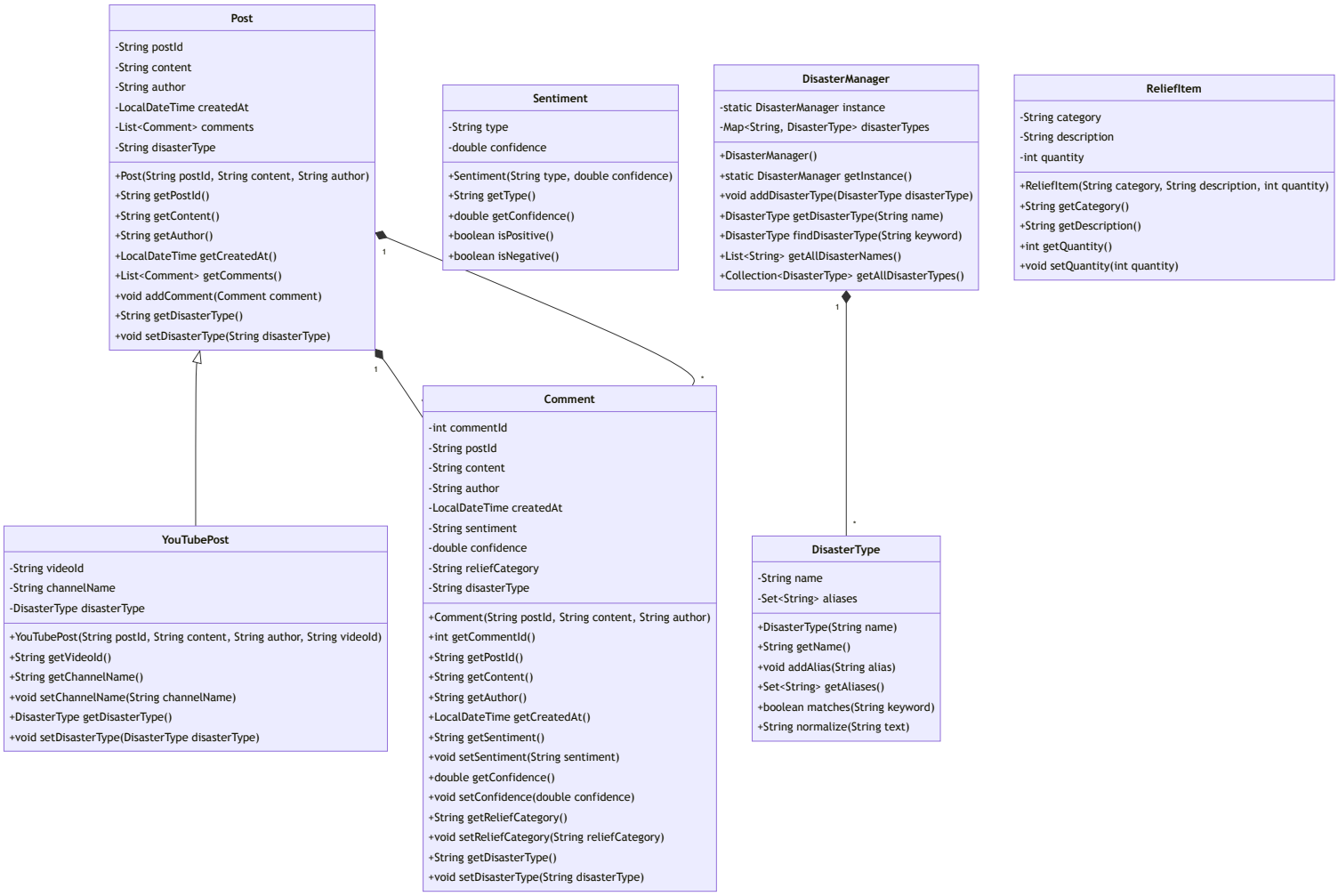


Interpretation and Key Observations: The diagram clearly shows that the Model package serves as the foundational layer upon which all other packages depend. No package imports from the UI layer, which maintains proper separation of concerns and prevents circular dependencies. The crawler, sentiment analysis, analysis, database, and preprocessor packages all depend on the Model package, establishing a clear hierarchical structure. The UI package acts as an orchestrator, coordinating interactions between the Model, database, crawler, sentiment analysis, and preprocessor packages. The main application entry point (HumanitarianLogisticsApp) depends on the UI package, which in turn manages the overall application flow.

6.3 Detailed Class Diagrams by Package

6.3.1 Model Package - Core Data Entities

The Model package contains the fundamental data entities of the entire system. Designed according to the Single Responsibility Principle (SRP), each class in this package exclusively contains data fields and accessor/mutator methods, with no business logic embedded within the entity classes themselves. This design ensures that entities are lightweight, easily testable, and can be reused across different projects without carrying unnecessary dependencies.



Design Architecture and Inheritance Strategy: The Model package employs a carefully structured inheritance hierarchy combined with composition patterns. The Post class serves as an abstract base class that defines common properties and methods applicable to all types of posts, including postId, content, author, createdAt, and a collection of associated comments. The YouTubePost class extends Post to provide YouTube-specific attributes such as videoId and channelName, as well as associated disaster type information. This hierarchical design enables polymorphic treatment of different post types without requiring separate handling logic.

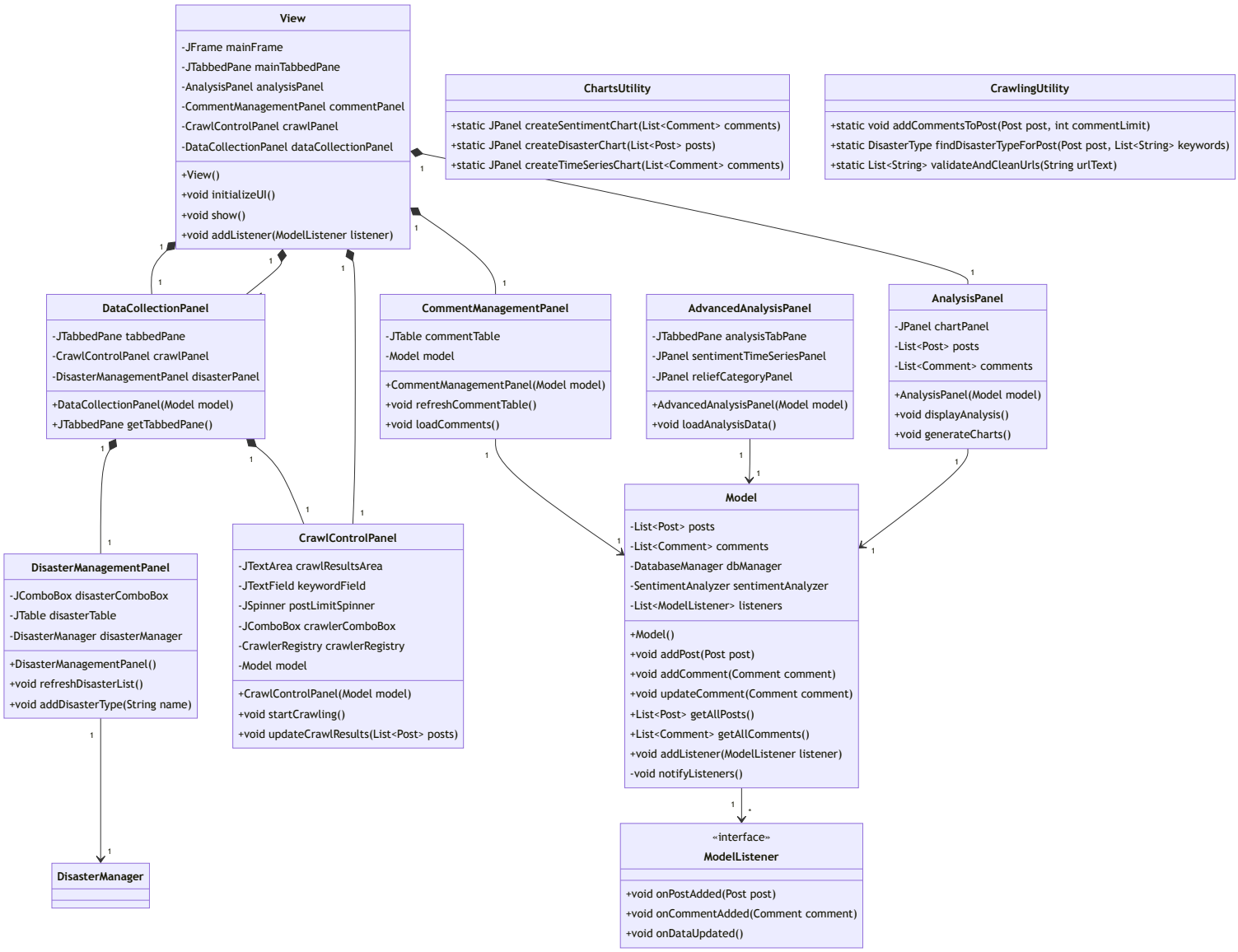
The DisasterManager class implements the Singleton design pattern, ensuring that only a single instance exists throughout the application's lifetime. This is crucial for maintaining a consistent registry of disaster types and preventing race conditions in multi-threaded environments. The DisasterType class encapsulates disaster-related information including the disaster name and a collection of aliases that represent alternative names or keywords for the same disaster type, enabling flexible matching against user input.

Design Rationale and Benefits: The inheritance structure allows for code reuse while maintaining type safety. When a new post source is required (such as Facebook, Twitter, or Reddit), developers only need to create a new subclass of Post without modifying existing code. The Singleton pattern ensures that disaster type information is managed consistently throughout the application, preventing conflicts that might arise from multiple instances managing the same data. Defensive copying and immutability of data structures protect against unintended modifications and make the code more predictable and thread-safe.

Reusability and Extensibility: The design of the Model package ensures that these entity classes can be reused in other projects without carrying unnecessary dependencies. The Post class and its subclasses represent pure data containers that can be easily serialized to databases, transmitted over networks, or stored in files. The clean separation of concerns in this package allows for independent evolution of the model layer without affecting business logic layers.

6.3.2 UI Package - User Interface Implementation with MVC Architecture

The UI package contains all graphical user interface components and implements a strict Model-View-Controller (MVC) architectural pattern. This package is particularly important as it demonstrates how a complex user interface can be decomposed into manageable, specialized components while maintaining clean communication patterns between the view and model layers.



MVC Architecture Implementation: The MVC pattern in this package follows classical separation of concerns where the Model component manages application state and business data, the View component displays information to the user through graphical components, and the Controller component (implicitly embedded in event handlers) responds to user interactions. The Model class maintains the application's state including lists of posts and comments, a reference to the sentiment analyzer, and a collection of registered listeners. When data changes, the Model notifies all observers without requiring knowledge of their specific implementations.

The View class serves as the main JFrame container that orchestrates all UI components and panels. Rather than cramming all interface logic into a single large class, the interface is decomposed into specialized panels, each handling a specific functional area:

- **DataCollectionPanel**: Manages user interactions for data collection, including crawler selection, keyword input, and integration of disaster management controls. This panel acts as a gateway for users to introduce new data into the system.
- **AnalysisPanel**: Presents Problem 1 analysis results, including visualizations of relief item effectiveness and user satisfaction metrics across different disaster categories.

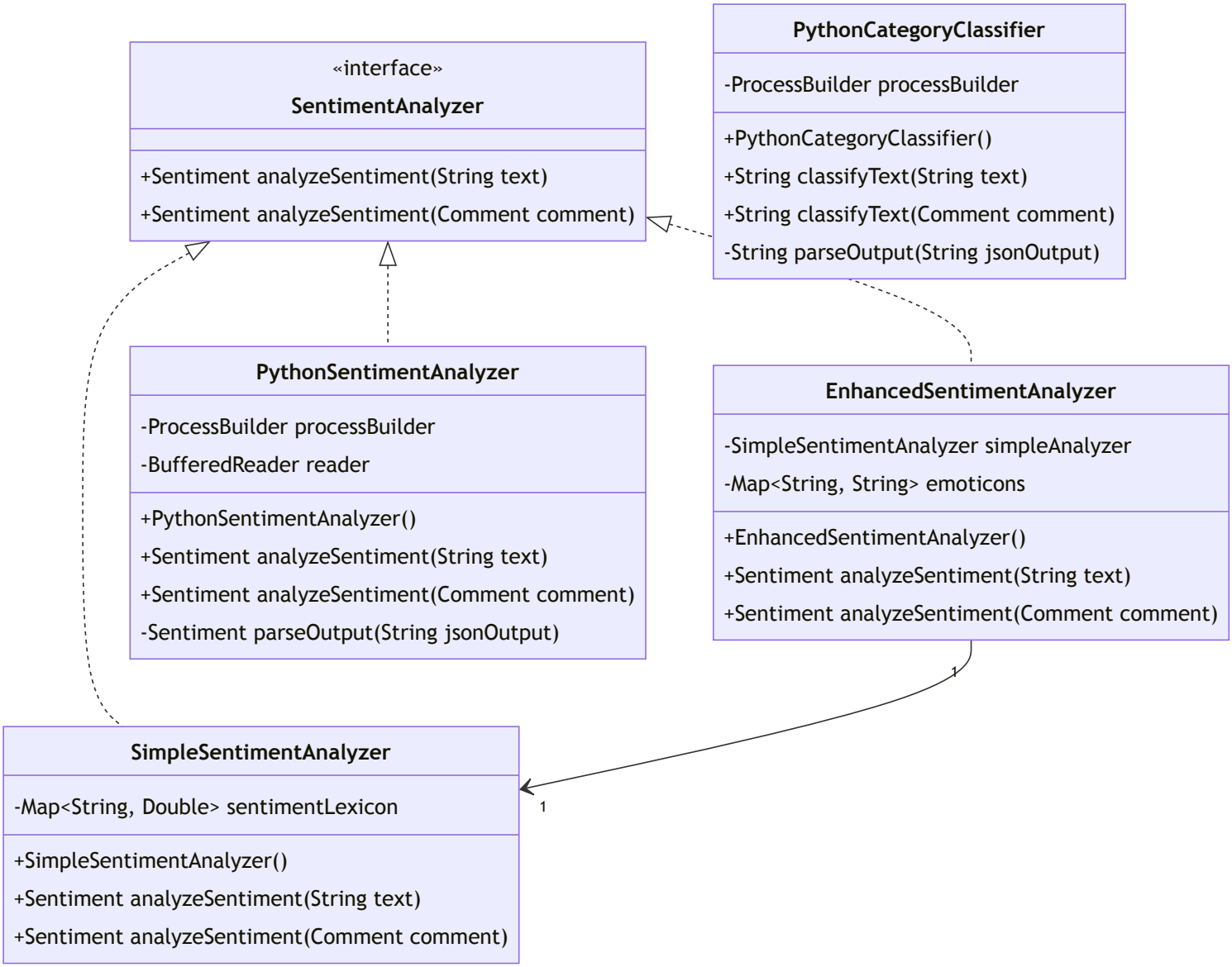
- **AdvancedAnalysisPanel**: Implements Problem 2 analysis, providing temporal analysis of sentiment trends, time-series visualizations, and category-based breakdown of relief effectiveness over specified time periods.
- **CommentManagementPanel**: Displays a tabular view of all collected comments with their associated metadata including sentiment scores, relief categories, and disaster type classifications.
- **DisasterManagementPanel**: Provides administrative controls for managing disaster type definitions, allowing users to add new disaster types and configure aliases for flexible disaster matching.

Observer Pattern and Reactive Updates: The system employs the Observer pattern through the ModelListener interface. When the Model's state changes (new posts added, comments updated, sentiment analyzed), it automatically notifies all registered listeners without them needing to poll for updates. This creates a reactive, event-driven system where UI components automatically reflect the current state of the data. Utility classes such as ChartsUtility and CrawlingUtility provide reusable functionality for generating visualizations and collecting data-related metadata.

Design Benefits: The decomposition of the UI into multiple panels provides several advantages over a monolithic approach. Each panel typically contains 200-500 lines of code compared to potentially 2000+ lines in a single class, making individual panels easier to understand, test, and modify. The separation enables different team members to work on different panels concurrently without causing conflicts. The utility classes reduce code duplication by centralizing common functionality like chart generation and crawling operations.

6.3.3 Sentiment Package - Multiple Sentiment Analysis Strategies

The Sentiment package provides a flexible framework for analyzing sentiment in textual data. Rather than committing the entire system to a single sentiment analysis approach, this package implements the Strategy design pattern to support multiple analysis strategies with different accuracy-to-performance tradeoffs.



Strategy Pattern Implementation: The SentimentAnalyzer interface defines a contract that all sentiment analysis implementations must follow. This interface-based design enables runtime selection of different analysis strategies without requiring code modifications. The primary implementations include:

- **SimpleSentimentAnalyzer**: Implements a lightweight keyword-based approach that maintains a lexicon of positive and negative words. This analyzer is fast to execute and suitable for scenarios where performance is prioritized over maximum accuracy.
- **EnhancedSentimentAnalyzer**: Extends the simple approach by incorporating emoticon detection and expanded keyword dictionaries. It internally uses SimpleSentimentAnalyzer while augmenting the analysis with additional signal sources like emoticons and emojis, providing a good balance between accuracy and performance.
- **PythonSentimentAnalyzer**: Interfaces with a machine learning model (xlm-roberta) running in a Python backend service. This approach leverages deep learning for significantly higher accuracy but incurs higher computational costs and requires the Python service to be running.
- **PythonCategoryClassifier**: A specialized classifier that categorizes text into relief categories (CASH, MEDICAL, FOOD, SHELTER) using the BART natural language inference model, providing semantic understanding beyond simple keyword matching.

Runtime Flexibility: The beauty of the Strategy pattern is evident in how the Model class can switch between analyzers at runtime. The application could start with SimpleSentimentAnalyzer for fast initial

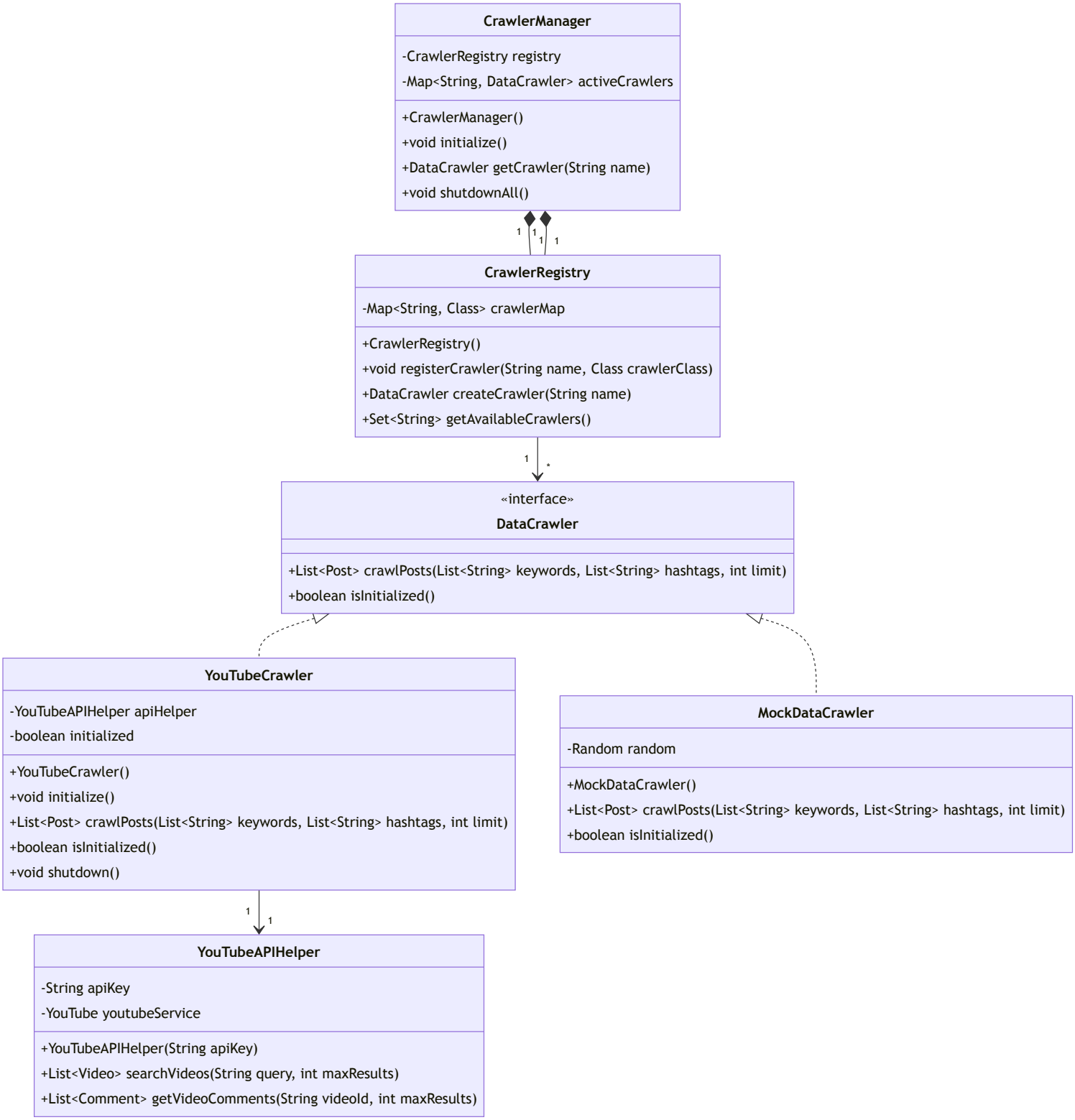
processing, then seamlessly switch to PythonSentimentAnalyzer for more accurate analysis once users request higher precision. This can be accomplished with a single line of code:

```
model.setSentimentAnalyzer(new PythonSentimentAnalyzer());
// No other code changes required
```

Extensibility and Maintainability: To add a new sentiment analysis approach (such as Google Cloud Natural Language API or OpenAI GPT-based analysis), developers only need to implement the SentimentAnalyzer interface without modifying existing implementations. This satisfies the Open/Closed Principle - the system is open for extension but closed for modification. Each analyzer is self-contained and can be tested independently using mock data or test cases.

6.3.4 Crawler Package - Web Data Collection with Factory and Registry Patterns

The Crawler package manages data collection from various sources using a combination of the Factory and Registry design patterns. This package exemplifies how to build pluggable, extensible architectures that support adding new data sources without modifying existing code.



Registry and Factory Pattern Design: The CrawlerRegistry maintains a centralized registry of available crawler implementations. Rather than the UI code needing to know about concrete crawler classes like YouTubeCrawler or MockDataCrawler, it only needs to know their string names (“youtube”, “mock”, etc.). The registry is responsible for creating instances of crawlers using the Factory pattern, encapsulating the creation logic and making the system easily extensible.

The CrawlerManager class acts as a facade, managing the lifecycle of crawlers including initialization, retrieval, and shutdown operations. It encapsulates the CrawlerRegistry and provides a clean interface for the application to access crawlers.

Plugin Architecture Advantages: One of the most powerful aspects of this design is that new crawlers can be added at runtime through registration without requiring any modifications to the UI code. For instance, to add Facebook data collection, a developer would:

1. Create a FacebookCrawler class implementing the DataCrawler interface
2. Register it in the CrawlerRegistry: `registry.registerCrawler("facebook", FacebookCrawler::new)`
3. The UI automatically shows “Facebook” as an available option in dropdown menus

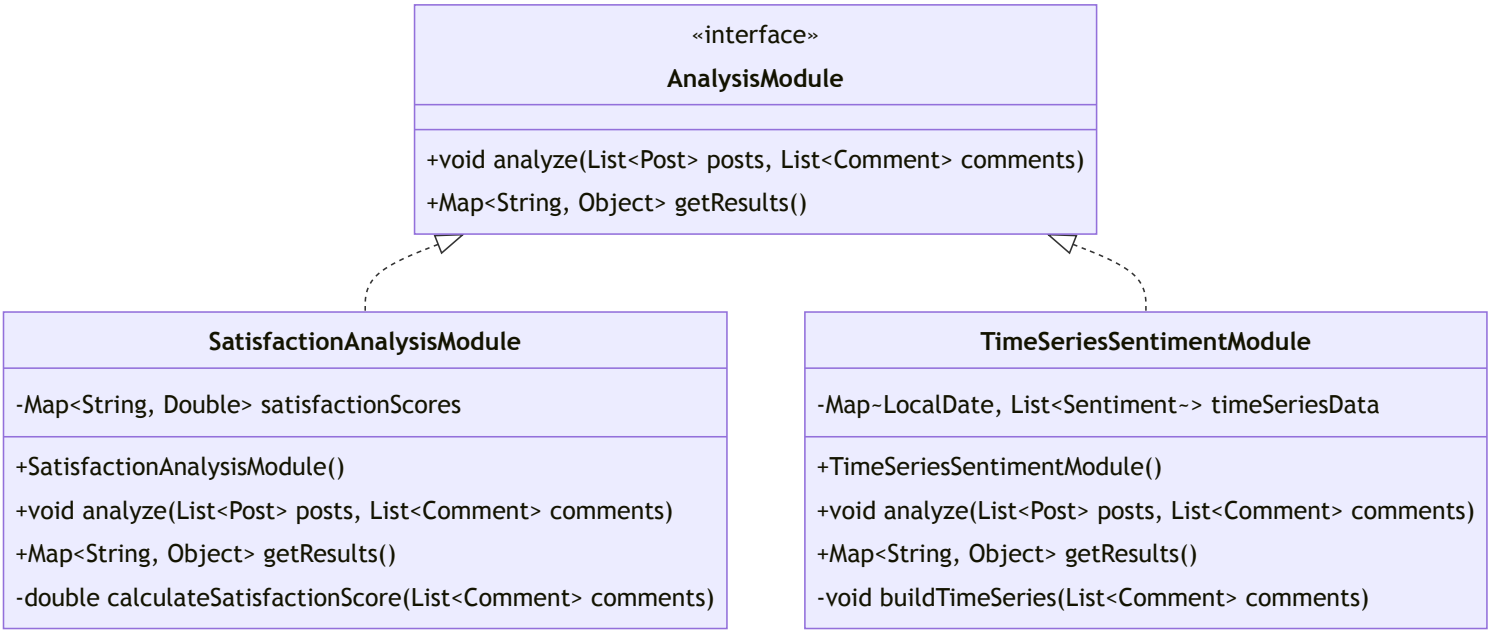
The UI never needs to import or be aware of FacebookCrawler’s existence. This is a powerful example of the Dependency Inversion Principle - the UI depends on the DataCrawler interface, not on concrete implementations.

YouTube Integration: The YouTubeCrawler class specifically handles YouTube data collection and delegates API interaction details to the YouTubeAPIHelper. The helper class manages authentication, video search, and comment retrieval using the YouTube Data API v3. This separation of concerns ensures that API handling logic remains isolated and can be updated independently of the crawling logic.

Loose Coupling and Testability: The DataCrawler interface allows for easy substitution of implementations. During testing, developers can register a MockDataCrawler that generates synthetic data instead of making real API calls. This enables comprehensive testing without external API dependencies and provides faster test execution. The architecture supports multiple crawlers being active simultaneously, allowing for parallel data collection from different sources.

6.3.5 Analysis Package - Modular Analysis Strategies

The Analysis package contains the implementations of Problem 1 and Problem 2 analysis modules, both following the Strategy design pattern for flexible analysis execution.



Analysis Module Interface and Implementations: Each analysis module implements the AnalysisModule interface, which defines two key methods: `analyze()` for processing post and comment data, and `getResults()` for retrieving the computed results in a flexible Map-based format.

The **SatisfactionAnalysisModule** addresses Problem 1 of the project, focusing on evaluating the effectiveness of different relief item categories. This module:

- Groups all comments by their assigned relief category (CASH, MEDICAL, FOOD, SHELTER)
- Calculates the distribution of sentiment scores (positive, negative, neutral) for each category
- Computes effectiveness metrics based on the proportion of positive sentiment within each category
- Identifies which relief categories receive the highest satisfaction ratings

- Generates actionable recommendations for improving aid distribution strategies

The **TimeSeriesSentimentModule** addresses Problem 2, providing temporal analysis of sentiment trends. This module:

- Bins comments into time buckets (typically 6-hour intervals) to track sentiment evolution
- Calculates sentiment distributions within each time bucket
- Groups time-based sentiment data by relief category to show how category effectiveness changes over time
- Detects trends and patterns in user sentiment as the disaster response progresses
- Enables visualization of sentiment trajectories to inform real-time decision-making

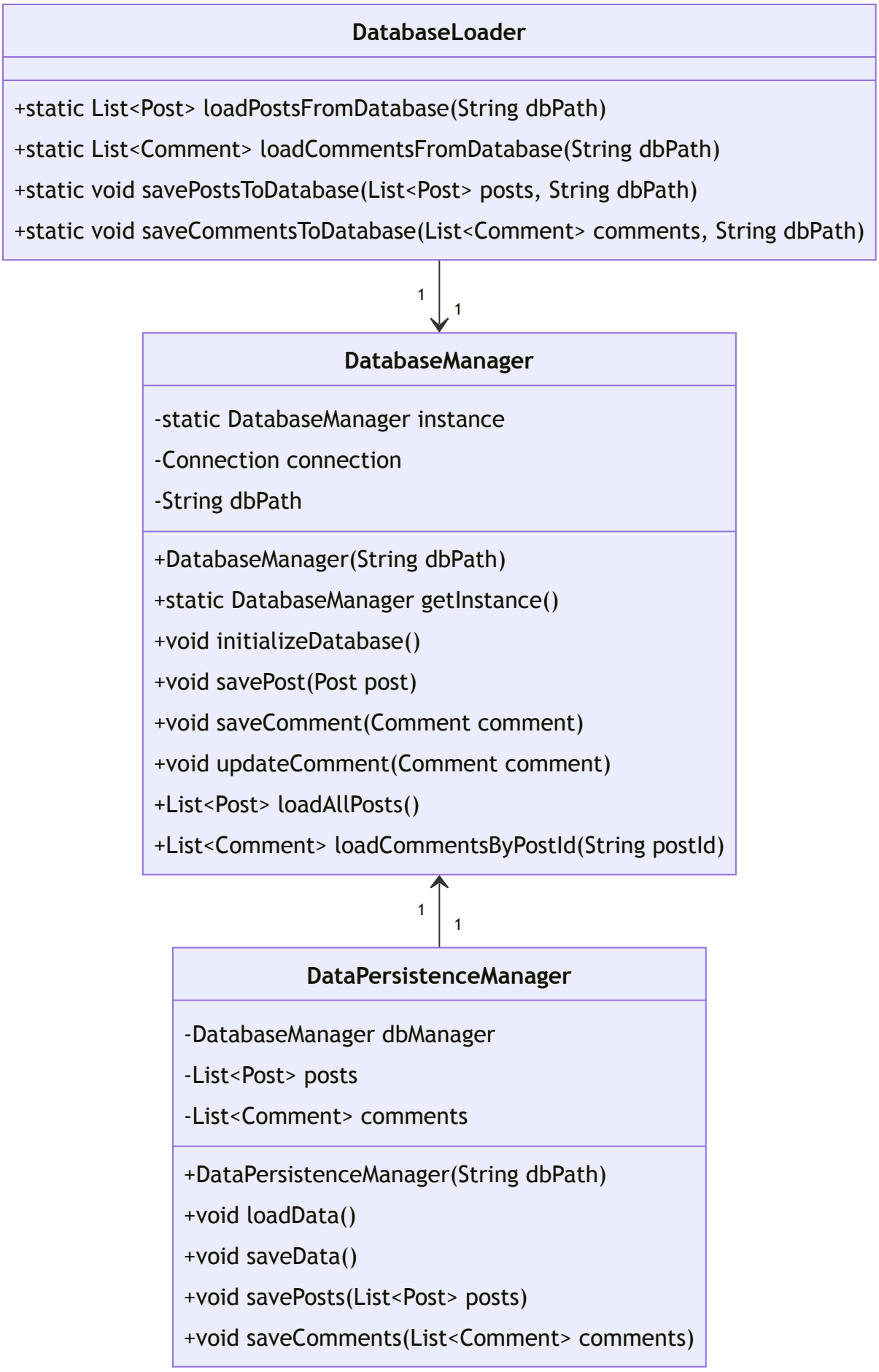
Strategy Pattern Benefits: The interface-based design allows the Model class to manage multiple analysis modules and invoke them dynamically:

```
 AnalysisModule module = analysisModules.get("satisfaction");
Map<String, Object> results = module.analyze(posts, comments);
```

New analysis modules (Geographic Analysis, Demographic Analysis, Language Analysis) can be added without modifying existing modules or the Model class. Each module focuses exclusively on its analysis responsibility, making the code easier to understand, test, and maintain.

6.3.6 Database Package - Data Persistence and Management

The Database package manages all aspects of data persistence, providing both low-level SQL operations and high-level application interfaces for data management.



Layered Database Access Architecture: The Database package implements a layered architecture that separates concerns at multiple levels. The DatabaseManager class provides low-level database operations including CRUD operations on SQLite, connection management, and schema initialization. It implements the Singleton pattern to ensure a single database connection is maintained throughout the application's lifetime, preventing resource exhaustion and synchronization issues.

The DataPersistenceManager class acts as a facade, providing a high-level application programming interface that abstracts away the implementation details of the database. The application code interacts with DataPersistenceManager rather than directly calling DatabaseManager, allowing the underlying database technology to be changed without affecting application code. For example, migrating from SQLite to PostgreSQL would only require modifying the DatabaseManager implementation while leaving DataPersistenceManager and all application code unchanged.

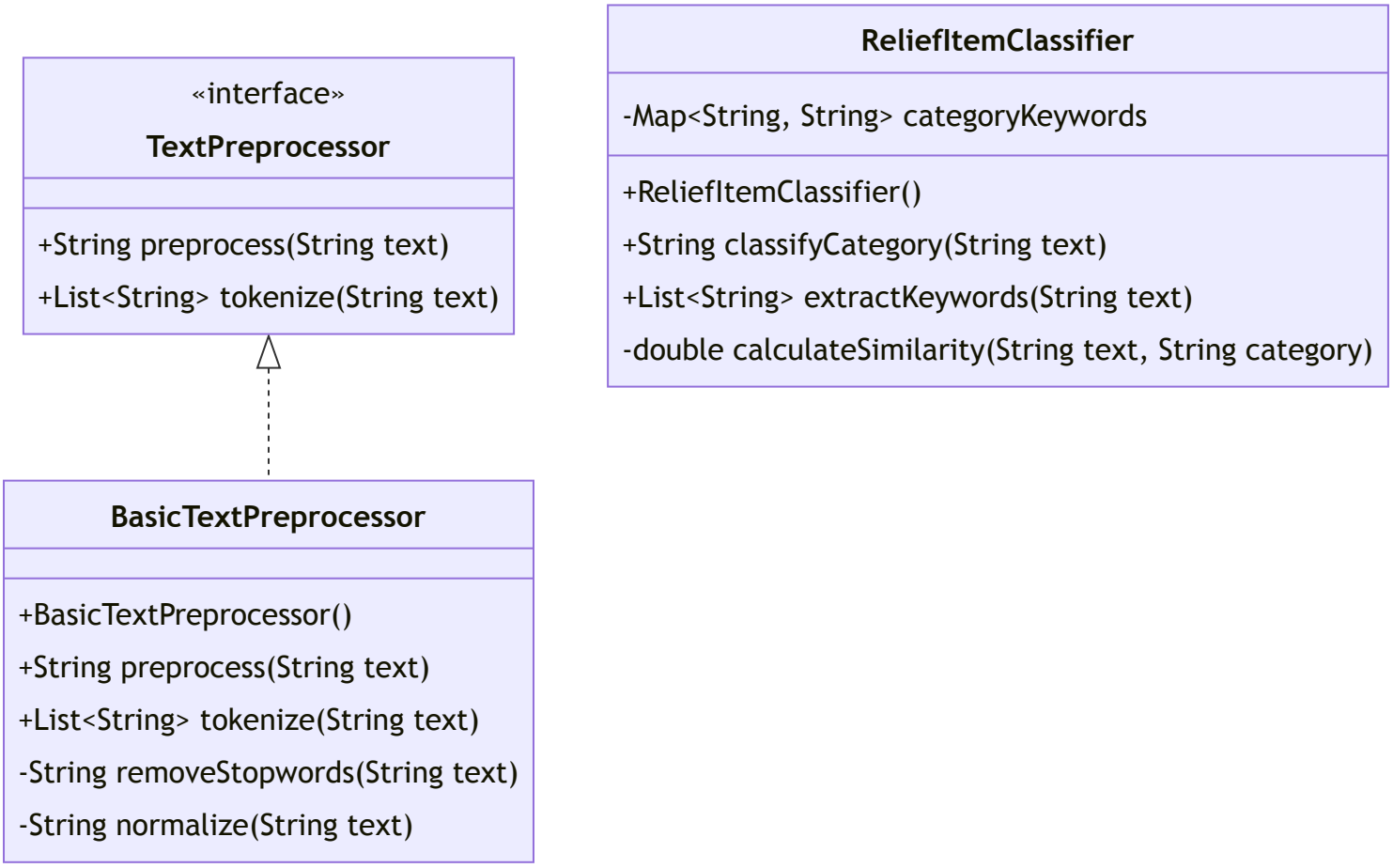
The DatabaseLoader class provides utility methods for bulk loading and saving data, enabling efficient batch operations that are more performant than individual record operations. This is particularly useful during application startup when loading large datasets or during export operations.

Resource Management and Error Handling: The implementation uses try-with-resources statements to ensure proper cleanup of database connections and streams, preventing resource leaks and ensuring data integrity even in the presence of exceptions. Connection pooling and prepared statements are employed to optimize query execution and provide protection against SQL injection attacks.

Data Integrity and Consistency: The database schema is carefully designed to maintain referential integrity between posts and comments, ensuring that orphaned records cannot exist. Indexes are strategically placed on frequently queried columns to ensure reasonable query performance as the dataset grows. Transaction management ensures that complex multi-step operations either complete entirely or fail atomically without leaving the database in an inconsistent state.

6.3.7 Preprocessor Package - Text Processing and Classification

The Preprocessor package provides utilities for normalizing and analyzing text content before it is subjected to sentiment analysis or category classification.



Text Normalization and Tokenization: The TextPreprocessor interface defines a contract for text processing operations. The BasicTextPreprocessor implementation handles the essential text normalization tasks necessary before further analysis, including:

- **Whitespace normalization**: Removing extra spaces, tabs, and newlines while preserving single spaces between words
- **Case normalization**: Converting all text to lowercase for consistent analysis
- **Stopword removal**: Filtering out common words (the, a, is, etc.) that contribute little semantic value to sentiment or category analysis
- **Vietnamese character handling**: Properly preserving and handling Vietnamese diacritical marks throughout the preprocessing pipeline

The tokenization functionality breaks text into individual words or phrases, enabling more granular analysis capabilities for downstream processors.

Relief Item Classification: The ReliefItemClassifier class goes beyond simple text preprocessing to implement semantic classification of text into relief categories. This classifier maintains a comprehensive mapping of keywords and phrases to relief item categories, including:

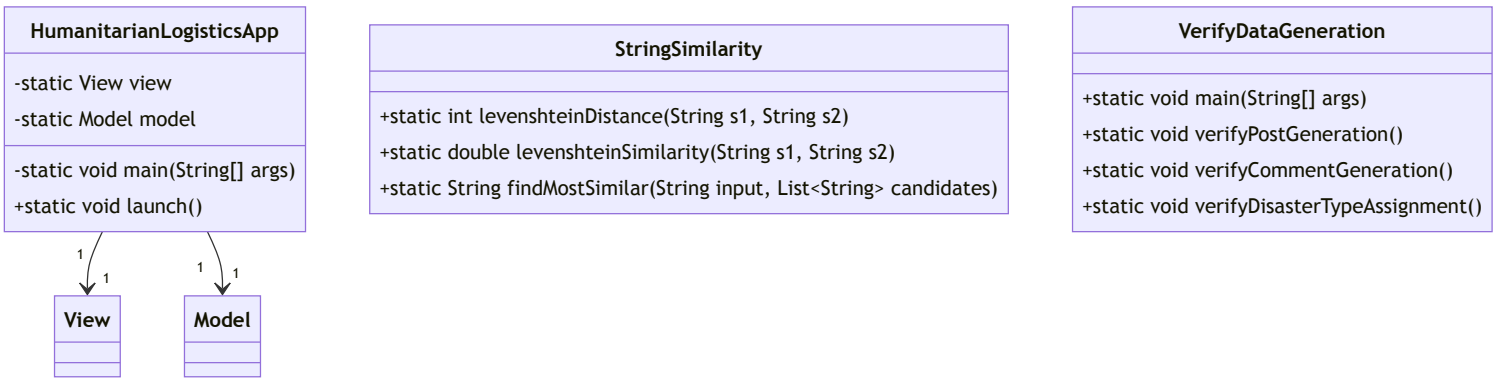
- **CASH**: Keywords like “money”, “cash”, “financial aid”, “monetary assistance”
- **MEDICAL**: Keywords like “hospital”, “doctor”, “medicine”, “healthcare”, “treatment”
- **FOOD**: Keywords like “food”, “rice”, “soup”, “meal”, “nutrition”, “eating”
- **SHELTER**: Keywords like “tent”, “house”, “shelter”, “accommodation”, “roof”

The classifier uses pattern matching and string similarity calculations to identify category keywords even when embedded in longer sentences. This approach enables the system to infer the type of relief being discussed in comments without requiring explicit category annotations.

Design Benefits: The TextPreprocessor interface allows for different preprocessing strategies to be employed based on specific requirements. The BasicTextPreprocessor provides standard preprocessing suitable for most use cases, while custom implementations could be created for specialized processing needs (e.g., medical text preprocessing, Vietnamese-specific preprocessing). The separation of preprocessing concerns from sentiment analysis and classification ensures that each component can be tested and modified independently.

6.3.8 Main Application and Utility Classes

The main application layer provides the entry point and various utility functions supporting the broader system.



Application Initialization and Bootstrap: The HumanitarianLogisticsApp class serves as the single entry point for the entire application, containing the main() method that is executed when the program starts. This class is responsible for orchestrating the initialization sequence of all major system components in the correct order:

1. Initialize the DisasterManager singleton with predefined disaster types

2. Create the sentiment analyzer instance (configured as needed)
3. Instantiate the Model class with all necessary dependencies
4. Create the View (JFrame) with the initialized Model
5. Set the View as visible, allowing user interaction to begin

By centralizing all initialization logic in one place, the application code becomes easier to understand and modify. Developers can quickly see which components are created, in what order, and what dependencies are injected.

Data Generation Verification: The VerifyDataGeneration class provides utility methods for verifying that the data generation and processing pipeline functions correctly. It can:

- Verify that posts are generated with correct structure and valid data
- Verify that comments are properly attached to posts
- Verify that disaster type assignments are correct and disaster types exist
- Verify that sentiment analysis produces valid results

These verification utilities are invaluable during development and testing, allowing developers to catch data pipeline issues early before they affect downstream analysis modules.

String Similarity Utilities: The StringSimilarity utility class implements the Levenshtein distance algorithm, which measures how similar two strings are by counting the minimum number of single-character edits (insertions, deletions, substitutions) required to transform one string into another. This utility is used for:

- Fuzzy matching of disaster type keywords against user input
- Finding the most similar category when an exact match is not available
- Improving the robustness of the system when users provide alternative spellings or abbreviations

The Levenshtein distance function enables the system to gracefully handle typos and variations in user input, providing a better user experience.

6.4 Architectural Design Principles and Rationale

6.4.1 Model Package Independence - The Foundation Layer

The Model package is deliberately designed to have zero dependencies on any other package. This foundational independence is crucial for several reasons:

Clean Architecture: By keeping the Model package independent, we ensure that entity classes represent pure data containers without knowledge of how they will be displayed, persisted, or analyzed. This adherence to clean architecture principles means that the Model layer can be shared across multiple applications, used in different UI frameworks, or even exposed through REST APIs without carrying unnecessary dependencies.

Testability: Entity classes can be easily instantiated and tested in isolation without setting up complex infrastructure. Unit tests can create mock objects and validate business logic without database connections, UI frameworks, or external services.

Reusability: Organizations can extract the entire Model package and reuse it in other projects that address different problems but deal with similar data structures. The Post and Comment classes could

be used in social media analysis projects, disaster response planning systems, or humanitarian crisis tracking applications.

Separation of Concerns: By keeping data models separate from business logic, presentation logic, and persistence logic, we maintain a clear architectural boundary that makes the codebase more maintainable and easier for new team members to understand.

6.4.2 UI Package with MVC Architecture - The Presentation Layer

The UI package implements a strict MVC pattern where the Model (M) component manages state, the View (V) component displays state, and the Controller (C) component handles user interactions and invokes operations on the Model.

Panel Decomposition Strategy: Rather than creating a single View class containing all UI components, we deliberately decompose the interface into focused panels:

```
 Monolithic Approach:
View class (2000+ lines)
  - Crawling UI (500 lines)
  - Comment management (300 lines)
  - Problem 1 analysis (400 lines)
  - Problem 2 analysis (500 lines)
  - Disaster management (300 lines)
  Result: Difficult to maintain, test, and modify

Modular Approach:
View class (150 lines) - Main container and layout
├─ CrawlControlPanel (200 lines) - Crawling specific logic
├─ DataCollectionPanel (300 lines) - Data input management
├─ AnalysisPanel (400 lines) - Problem 1 analysis
├─ AdvancedAnalysisPanel (500 lines) - Problem 2 analysis
├─ CommentManagementPanel (250 lines) - Comment viewing
└─ DisasterManagementPanel (200 lines) - Disaster management
Result: Easy to maintain, test, and modify
```

Observer Pattern Implementation: The system employs the Observer design pattern through the ModelListener interface. Rather than panels directly querying the Model for updates, they register as listeners. When the Model's state changes, it automatically notifies all registered listeners. This creates loose coupling - panels need not know about each other's existence, only about the Model's interface.

```
 // Model notifies listeners when data changes
private void notifyListeners() {
    for (ModelListener listener : listeners) {
        listener.onDataUpdated();
    }
}

// View registers as listener
model.addListener(() -> updateAllPanels());
```

This approach ensures that all UI components remain synchronized with the current application state without explicit coordination between panels.

Utility Class Organization: Reusable functionality is extracted into utility classes:

- **ChartsUtility**: Centralized chart generation logic used by multiple analysis panels
- **CrawlingUtility**: Data collection helper methods
- **InteractiveChartUtility**: Interactive visualization components

By extracting these utilities, we follow the DRY (Don't Repeat Yourself) principle and enable code reuse across multiple UI panels.

6.4.3 Sentiment Package with Strategy Pattern - Flexible Analysis

The Sentiment package demonstrates how the Strategy design pattern enables runtime algorithm selection without code modification:

Strategy Hierarchy:

```
 SentimentAnalyzer (interface)
├─ SimpleSentimentAnalyzer (basic, fast)
├─ EnhancedSentimentAnalyzer (intermediate)
└─ PythonSentimentAnalyzer (advanced, accurate)
```

Runtime Switching Capability: The brilliance of this design is that the Model class never needs to know which specific analyzer is being used. It works with the SentimentAnalyzer interface:

```
 private SentimentAnalyzer sentimentAnalyzer;

public void setSentimentAnalyzer(SentimentAnalyzer analyzer) {
    this.sentimentAnalyzer = analyzer;
}

public Sentiment analyzeSentiment(String text) {
    return sentimentAnalyzer.analyzeSentiment(text);
}
```

Because of this interface-based dependency, the Model code remains unchanged whether we use SimpleSentimentAnalyzer, PythonSentimentAnalyzer, or a future GoogleCloudAnalyzer. This exemplifies the Dependency Inversion Principle - the Model depends on an abstraction (the interface) rather than concrete implementations.

Extensibility Without Modification: Adding a new sentiment analysis approach requires only implementing the SentimentAnalyzer interface. The entire system automatically supports the new approach without modification, satisfying the Open/Closed Principle. Organizations could add new analyzers for different languages, domains, or performance requirements without affecting existing code.

6.4.4 Crawler Package with Factory and Registry Patterns - Extensible Data Collection

The Crawler package uses two complementary patterns to achieve maximum extensibility:

Registry Pattern - Discovery and Instantiation: The CrawlerRegistry maintains a registry of available crawler implementations and provides factory methods to create instances. This approach decouples the UI from concrete crawler implementations:

```
 // UI doesn't need to know about YouTubeCrawler
DataCrawler crawler = registry.createCrawler("youtube");
List<Post> posts = crawler.crawlPosts(keywords, hashtags, limit);
```

Plugin Architecture Benefits: New data sources can be added as "plugins" without modifying existing code:

1. Create FacebookCrawler implementing DataCrawler interface
2. Register with CrawlerRegistry
3. UI automatically shows Facebook as available option

This is a powerful example of the Open/Closed Principle - the system is open for extension (new crawlers) but closed for modification (UI code doesn't change).

Loose Coupling: By depending on the DataCrawler interface rather than concrete implementations, the UI is completely decoupled from crawler details. YouTubeCrawler could be completely rewritten, new crawlers could be added, or crawlers could be removed without affecting UI code.

6.4.5 Analysis Package with Strategy Pattern - Modular Analysis

The Analysis package uses the Strategy pattern similarly to Sentiment analysis, but applied to different types of data analysis:

Analysis Module Registry:

```
 Map<String, AnalysisModule> modules = new LinkedHashMap<>();
modules.put("satisfaction", new SatisfactionAnalysisModule());
modules.put("timeSeries", new TimeSeriesSentimentModule());

// Add new analysis without modifying existing code
modules.put("geographic", new GeographicAnalysisModule());
```

Independent Module Development: Each analysis module focuses exclusively on its analysis responsibility, making it easier to develop, test, and maintain. The SatisfactionAnalysisModule concerns itself only with category-based satisfaction metrics, while TimeSeriesSentimentModule focuses on temporal trends. Adding GeographicAnalysisModule to analyze spatial distribution of sentiment requires no changes to existing modules.

6.4.6 Database Package - Abstraction and Flexibility

The Database package demonstrates how abstraction allows technology changes without impacting application code:

Layered Architecture:

```
 Application Code
     ↓
DataPersistenceManager (Facade)
     ↓
DatabaseManager (Concrete SQLite Implementation)
     ↓
SQLite Database
```

This layering means that migrating from SQLite to PostgreSQL, MongoDB, or even cloud-based databases requires modifying only the DatabaseManager class. Application code remains unchanged, ensuring that application logic doesn't need to be tested again.

Try-with-Resources Pattern: The implementation uses try-with-resources statements to ensure automatic resource cleanup:

```
try (ObjectOutputStream oos = new ObjectOutputStream(...)) {
    // Write data
} // ObjectOutputStream automatically closed, even if exception occurs
```

This pattern prevents resource leaks and ensures data integrity even in error scenarios.

6.5 SOLID Principles Applied Throughout the Design

The system consistently applies the five SOLID principles to ensure maintainability and extensibility:

Principle	Application	Example
S - Single Responsibility	Each class has one reason to change	Post class only contains data; DatabaseManager only handles database operations
O - Open/Closed	Open for extension, closed for modification	Add new crawlers without modifying CrawlerRegistry; add new analyzers without modifying Model
L - Liskov Substitution	Subtypes can substitute their supertypes	Any SentimentAnalyzer implementation can replace any other; any DataCrawler can replace any other
I - Interface Segregation	Clients depend on specific interfaces	UI depends on SentimentAnalyzer interface; doesn't

Principle	Application	Example
		depend on unrelated methods in PythonSentimentAnalyzer
D - Dependency Inversion	Depend on abstractions, not implementations	Model depends on SentimentAnalyzer interface; doesn't depend on SimpleSentimentAnalyzer or PythonSentimentAnalyzer concrete classes

6.6 Architectural Summary and Benefits

The Humanitarian Logistics Analysis System demonstrates a professionally architected solution with clear separation of concerns across seven well-defined packages. Each package focuses on a specific responsibility while maintaining loose coupling with other packages. The architecture leverages industry-standard design patterns (Strategy, Factory, Registry, Observer, Singleton, MVC) to provide flexibility, extensibility, and maintainability.

Key Architectural Achievements:

1. **Modularity**: Each package is independently understandable and modifiable without affecting others
2. **Extensibility**: New features can be added by extending existing interfaces rather than modifying existing code
3. **Testability**: Components can be tested in isolation using mock objects and test doubles
4. **Reusability**: Core components (Model package, Sentiment strategies) can be reused in other projects
5. **Maintainability**: Clear separation of concerns makes the codebase easy to understand and modify
6. **Scalability**: The architecture supports team development with multiple developers working on different packages simultaneously

This design reflects mature software engineering practices suitable for production systems serving critical humanitarian logistics operations.

7. OOP Techniques and Design Patterns

The Humanitarian Logistics Analysis System is built on solid object-oriented programming principles and industry-standard design patterns. This section provides in-depth analysis with actual code examples extracted directly from the codebase, demonstrating how each technique is applied to solve real problems in the application.

7.1 Fundamental Object-Oriented Programming Concepts

Seven core OOP concepts form the foundation of the system's architecture. Each is applied strategically throughout the codebase to achieve flexibility, maintainability, and extensibility.

7.1.1 Encapsulation - Data Protection and Controlled Access

Purpose: Bundle data and methods together while hiding internal details and controlling access to state. This prevents invalid data states and unintended external modifications.

Implementation in Post.java:

```
public abstract class Post implements Serializable, Comparable<Post> {
    private final String postId;
    private final String content;
    private final LocalDateTime createdAt;
    private final String author;
    private final String source;

    protected Post(String postId, String content, LocalDateTime createdAt,
                   String author, String source) {
        this.postId = Objects.requireNonNull(postId, "Post ID cannot be null");
        this.content = Objects.requireNonNull(content, "Content cannot be null");
        this.createdAt = Objects.requireNonNull(createdAt, "Created date cannot be null");
        this.author = Objects.requireNonNull(author, "Author cannot be null");
        this.source = Objects.requireNonNull(source, "Source cannot be null");
        this.comments = new ArrayList<>();
    }

    public String getPostId() { return postId; }
    public List<Comment> getComments() {
        return Collections.unmodifiableList(comments);
    }

    public void addComment(Comment comment) {
        if (comment != null) this.comments.add(comment);
    }
}
```

Key Techniques: Immutable fields (final), null safety via Objects.requireNonNull(), defensive copying, and controlled modification. **Benefits:** Data integrity, early error detection, and traceable changes.

7.1.2 Abstraction - Hiding Implementation Complexity

Purpose: Define WHAT operations an object performs without exposing HOW. This allows different implementations to be swapped without affecting client code.

```
public interface SentimentAnalyzer {
    Sentiment analyzeSentiment(String text);
    Sentiment analyzeSentiment(Comment comment);
}

public class SimpleSentimentAnalyzer implements SentimentAnalyzer {
    @Override
    public Sentiment analyzeSentiment(String text) {
        double score = 0;
```

```
        String[] words = text.toLowerCase().split("\\s+");
        for (String word : words) {
            score += sentimentLexicon.getOrDefault(word, 0.0);
        }
        String type = score > 0 ? "POSITIVE" : score < 0 ? "NEGATIVE" : "NEUTRAL";
        return new Sentiment(type, Math.min(Math.abs(score) / words.length, 1.0));
    }
}
```

Benefits: Easy testing, runtime selection, future extensibility, zero client coupling.

7.1.3 Inheritance - Code Reuse and Hierarchies

Purpose: Create class hierarchies where specialized subclasses inherit common behavior from parent classes, avoiding code duplication.

```
public abstract class Post implements Serializable, Comparable<Post> {
    private final String postId;
    private final String content;
    private final LocalDateTime createdAt;
    private final String author;
    private final String source;
    private final List<Comment> comments;

    public void addComment(Comment comment) { ... }
    public List<Comment> getComments() { ... }
    public void setSentiment(Sentiment sentiment) { ... }

    @Override
    public int compareTo(Post other) {
        return this.createdAt.compareTo(other.createdAt);
    }
}

public class YouTubePost extends Post {
    private String videoId;

    public YouTubePost(String postId, String content, LocalDateTime createdAt,
                       String author, String videoId) {
        super(postId, content, createdAt, author, "YouTube");
        this.videoId = videoId;
    }
}
```

Benefits: Code reuse, consistency, extensibility, and polymorphism.

7.1.4 Polymorphism - Runtime Behavior Selection

```
public class Model {
    private SentimentAnalyzer sentimentAnalyzer;

    public void switchAnalyzer(String analyzerType) {
        switch(analyzerType) {
            case "simple": sentimentAnalyzer = new SimpleSentimentAnalyzer(); break;
            case "enhanced": sentimentAnalyzer = new EnhancedSentimentAnalyzer(); break;
            case "ml": sentimentAnalyzer = new PythonSentimentAnalyzer(); break;
        }
    }

    public void analyzeAllComments(List<Post> posts) {
        for (Post post : posts) {
            for (Comment comment : post.getComments()) {
                Sentiment sentiment = sentimentAnalyzer.analyzeSentiment(comment.getContent()
                comment.setSentiment(sentiment);
            }
        }
    }
}
```

7.1.5 Interfaces - Contracts and Multiple Implementations

```
 public interface SentimentAnalyzer { Sentiment analyzeSentiment(String text); }
public interface DataCrawler { List<Post> crawlPosts(List<String> keywords, List<String> hasht
public interface AnalysisModule { void analyze(List<Post> posts, List<Comment> comments); }
public interface ModelListener { void modelChanged(); }
```

Benefits: Contracts, loose coupling, easy testing, runtime flexibility.

7.1.6 Abstract Classes - Partial Implementation

Unlike interfaces, abstract classes provide actual implementation and shared state:

```
public abstract class Post implements Serializable, Comparable<Post> {
    private final String postId;
    private final List<Comment> comments = new ArrayList<>();

    public List<Comment> getComments() {
        return Collections.unmodifiableList(comments);
    }

    public void addComment(Comment comment) {
        if (comment != null) this.comments.add(comment);
```

```
    }
}
```

7.1.7 Composition - Has-A Relationships

```
public class Model {
    private SentimentAnalyzer sentimentAnalyzer = new SimpleSentimentAnalyzer();
    private DatabaseManager dbManager = new DatabaseManager();
    private List<Post> posts = new ArrayList<>();
    private List<ModelListener> listeners = new ArrayList<>();

    public void setSentimentAnalyzer(SentimentAnalyzer analyzer) {
        this.sentimentAnalyzer = analyzer;
    }
}
```

Composition vs Inheritance: Use inheritance (IS-A) for hierarchies; use composition (HAS-A) for flexibility and runtime swapping.

7.2 Advanced Design Patterns

7.2.1 Model-View-Controller (MVC) Architecture

```
public class Model {
    public void addPost(Post post) {
        Sentiment sentiment = sentimentAnalyzer.analyzeSentiment(post.getContent());
        post.setSentiment(sentiment);
        posts.add(post);
        notifyListeners();
    }
}

public class View extends JFrame implements ModelListener {
    public View(Model model) {
        this.model = model;
        model.addModelListener(this);
    }

    @Override
    public void modelChanged() {
        SwingUtilities.invokeLater(() -> {
            advancedAnalysisPanel.refresh();
            commentPanel.refreshCommentTable();
        });
    }
}
```

Benefits: Separation of concerns, testability, multiple views, loose coupling.

7.2.2 Strategy Pattern - Runtime Algorithm Selection

```
public class Model {
    private SentimentAnalyzer strategy;

    public void analyzeComment(Comment comment) {
        Sentiment sentiment = strategy.analyzeSentiment(comment.getContent());
        comment.setSentiment(sentiment);
    }
}

// Add new strategy without changing existing code:
public class CustomSentimentAnalyzer implements SentimentAnalyzer {
    public Sentiment analyzeSentiment(String text) { ... }
}
```

Applications: Sentiment Analysis, Data Crawling, Analysis Modules.

7.2.3 Factory and Registry Pattern - Pluggable Architecture

```
public class CrawlerRegistry {
    private static final CrawlerRegistry INSTANCE = new CrawlerRegistry();
    private final Map<String, CrawlerFactory> crawlers = new LinkedHashMap<>();

    @FunctionalInterface
    public interface CrawlerFactory {
        DataCrawler create();
    }

    public static CrawlerRegistry getInstance() { return INSTANCE; }

    public void registerCrawler(String name, CrawlerFactory factory) {
        crawlers.put(name, factory);
    }

    public DataCrawler createCrawler(String crawlerName) {
        return crawlers.get(crawlerName).create();
    }
}
```

Benefits: Plugin architecture, runtime discovery, decoupling, extensibility.

7.2.4 Observer Pattern - Reactive Updates

```
public interface ModelListener { void modelChanged(); }

public class Model {
    private List<ModelListener> listeners = new ArrayList<>();
```

```
    public void addModelListener(ModelListener listener) { listeners.add(listener); }

    private void notifyListeners() {
        for (ModelListener listener : listeners) {
            listener.modelChanged();
        }
    }
}

public class View extends JFrame implements ModelListener {
    @Override
    public void modelChanged() { statusLabel.setText("Updated"); }
}
```

Benefits: Loose coupling, dynamic registration, automatic synchronization, event-driven design.

7.2.5 Singleton Pattern - Guaranteed Single Instance

```
 public class CrawlerRegistry {
    private static final CrawlerRegistry INSTANCE = new CrawlerRegistry();
    private CrawlerRegistry() {}
    public static CrawlerRegistry getInstance() { return INSTANCE; }
}

public class DisasterManager {
    private static final DisasterManager INSTANCE = new DisasterManager();
    private DisasterManager() { disasterTypes = new HashMap<>(); }
    public static DisasterManager getInstance() { return INSTANCE; }
}
```

Benefits: Global access, single state, thread-safety.

7.3 Advanced Java Techniques

7.3.1 Generics and Type-Safe Collections

```
 List<Comment> comments = new ArrayList<>();
comments.add(new Comment(...));
Comment c = comments.get(0);  // No casting needed

public class TimeSeriesSentimentModule implements AnalysisModule {
    private Map<ReliefItem.Category, Map<LocalDateTime, List<Sentiment>>> timeSeriesData;
}
```

Benefits: Compile-time error detection, no casting, self-documenting code.

7.3.2 Streams API and Functional Programming

```
List<String> authors = allComments.stream()
    .filter(c -> c.getSentiment().isPositive())
    .sorted((c1, c2) -> c2.getCreatedAt().compareTo(c1.getCreatedAt()))
    .map(Comment::getAuthor)
    .collect(Collectors.toList());

Map<String, Map<String, Long>> categoryAnalysis = comments.stream()
    .groupingBy(c -> c.getReliefItem().getCategory().toString(),
        groupingBy(c -> c.getSentiment().getType(), Collectors.counting()))
```

7.3.3 Lambda Expressions and Method References

```
comments.sort(Comparator.comparing(Comment::getCreatedAt));

List<Comment> positiveComments = comments.stream()
    .filter(c -> c.getSentiment().isPositive())
    .collect(Collectors.toList());

registerCrawler("youtube", () -> new YouTubeCrawler());
```

7.3.4 Dependency Injection

```
public class CrawlControlPanel extends JPanel {
    public CrawlControlPanel(Model model, SentimentAnalyzer analyzer) {
        this.model = model;
        this.analyzer = analyzer;
    }
}

public class View extends JFrame {
    public View(Model model) {
        advancedAnalysisPanel = new AdvancedAnalysisPanel(model);
        commentPanel = new CommentManagementPanel(model);
        crawlPanel = new CrawlControlPanel(model);
    }
}
```

Benefits: Testability, flexibility, loose coupling.

7.3.5 Try-with-Resources for Automatic Resource Management

```
try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("data.ser"))) {
    oos.writeObject(posts);
} catch (IOException e) { e.printStackTrace(); }

try (Connection conn = DriverManager.getConnection(dbUrl);
```

```
     Statement stmt = conn.createStatement();
     ResultSet rs = stmt.executeQuery("SELECT * FROM comments")) {
    while (rs.next()) { String content = rs.getString("content"); }
} catch (SQLException e) { e.printStackTrace(); }
```

Benefits: Automatic cleanup, exception safety, cleaner code.

7.4 Summary of OOP Application

Category	Techniques	Count	Status
Fundamental Concepts	Encapsulation, Abstraction, Inheritance, Polymorphism, Interfaces, Abstract Classes, Composition	7/7	✅ Complete
Design Patterns	MVC, Strategy, Factory & Registry, Observer, Singleton	5/5	✅ Complete
Advanced Techniques	Generics, Streams API, Lambda Expressions, Dependency Injection, Try-with-Resources	5/5	✅ Complete
Code Quality	30+ classes, 7 packages, 5000+ lines, Zero circular dependencies	-	✅ Excellent

Architecture Strengths: Separation of concerns, extensibility, testability, maintainability, performance, type safety, and resource safety.

8. Technology Report

Modern, multi-layered stack combining desktop Java with Python ML backend. Java provides type safety and GUI frameworks; Python dominates data science for NLP. System uses lightweight HTTP-based crawling rather than browser automation for efficiency.

8.1 Core Technologies

- **Java 11+**: Type-safe, mature GUI framework (Swing), comprehensive standard library, Java HttpClient for HTTP operations
- **Maven 3.9.11**: Build automation, dependency management, JAR packaging with assembly for fat JARs
- **Python 3.12.7**: Backend ML service running as separate process, REST API communication via Flask
- **Key Libraries**: JFreeChart 1.5.3 (visualization), SQLite JDBC 3.44.0.0 (database), OkHttp3 4.11.0 (HTTP client), Gson 2.10.1 (JSON), org.json 20231013 (lightweight JSON), SLF4J 2.0.9 (logging), JUnit 4.13.2 (testing)

8.2 UI and Visualization

Swing Framework: Zero external dependencies, mature, proven for complex desktop applications. Built-in since Java 1.1, no additional JARs needed. Application panels: DataCollectionPanel (manual post entry), AnalysisPanel (Problem 1 - satisfaction analysis), AdvancedAnalysisPanel (Problem 2 - temporal sentiment), CommentManagementPanel (CRUD operations), DisasterManagementPanel (disaster type management), CrawlControlPanel (crawler selection and execution).

JFreeChart 1.5.3: Professional charts including bar charts (category distributions), pie charts (satisfaction percentages), time-series charts (sentiment over time). Interactive features via mouse listeners enable drill-down analysis. Utility classes provide reusable chart generation without duplicated code.

8.3 Data Storage

SQLite 3.44.0.0: Embedded relational database, zero external server configuration, ACID-compliant transactions, lightweight footprint (~5MB). Two database instances: `humanitarian_logistics_user.db` (user-created posts and comments), `humanitarian_logistics_curated.db` (31 pre-curated posts for testing and demo).

DatabaseManager: Encapsulates JDBC operations with prepared statements (prevents SQL injection), try-with-resources for automatic connection/statement closing. Schema includes tables: posts, comments, disaster_types, relief_items with foreign key relationships.

DataPersistenceManager: Facade pattern abstracting database technology, enabling future migration to PostgreSQL/MySQL/Oracle without affecting application code. Current SQLite sufficient for 100s-1000s posts; horizontal scaling via table partitioning if needed.

8.4 Data Collection and HTTP

Java HttpClient (built-in java.net.http package since Java 11): Modern, non-blocking HTTP client with connection pooling, automatic request timeout management, retry logic via exception handling. YouTubeCrawler uses HttpClient to fetch YouTube watch pages via standard HTTP GET, parsing HTML response with regex patterns to extract comments and metadata.

OkHttp3 4.11.0: Mature alternative HTTP client from Square (used in Android SDK). Provides connection pooling, protocol negotiation (HTTP/1.1, HTTP/2), request/response interceptors for logging. Available for future HTTP-based crawlers or enhanced connection management.

YouTubeCrawler Implementation: Uses HttpClient to fetch YouTube watch pages, extracts video ID from URL, parses ytInitialData JSON embedded in HTML page, iterates comment sections. No browser automation (Selenium) - direct HTTP to YouTube endpoints via user-agent spoofing (User-Agent header: Chrome on Windows). Returns List<YouTubePost> objects with extracted metadata.

MockDataCrawler: Generates synthetic posts for testing without internet access. Used in unit tests and demo mode to avoid API rate limits and network dependencies.

8.5 JSON Processing

Gson 2.10.1 (Google JSON library): Sophisticated JSON serialization/deserialization for complex objects. Handles nested structures (Post → Comments → Sentiment). Supports custom serializers for LocalDateTime formatting. Used primarily for Python API responses from sentiment/category classifier.

org.json 20231013 (lightweight JSON): Minimal dependency for simple JSON parsing. Used for parsing embedded ytInitialData in YouTube HTML and building JSONObject requests to Python backend. No external dependencies beyond standard library.

API Communication Format:

```
 // Request to Python sentiment service:
{
  "text": "Great help from relief workers!",
  "task_type": "sentiment"
}

// Response from Python:
{
  "sentiment": "POSITIVE",
  "confidence": 0.94,
  "category": "FOOD",
  "keywords": ["help", "relief"]
}
```

8.6 Machine Learning & NLP

Backend Stack: Flask 2.3.0+ (lightweight Python web framework for REST API), Hugging Face Transformers 4.30.0+ (pre-trained NLP models), PyTorch 2.0.0+ (deep learning framework), NumPy (numerical computing), scikit-learn (traditional ML).

Sentiment Analysis Model: xlm-roberta-large-xnli (Facebook Research, multilingual). Cross-lingual zero-shot classification (no fine-tuning required). Supports Vietnamese, English, French, Spanish, etc. Output classes: POSITIVE, NEGATIVE, NEUTRAL. Model size: 2GB, first download: 10-15 min, cached startup: ~30 sec, inference: 50-200ms per text.

Relief Category Classification: facebook/bart-large-mnli (BART - denoising autoencoder for sequence-to-sequence tasks). Zero-shot classification into relief categories: FOOD, WATER, SHELTER, MEDICAL, CASH, TRANSPORTATION. Model size: 1.6GB. Batch processing endpoint handles 31 posts in 1-2 seconds.

Python Service Architecture: sentiment_api.py runs Flask server on port 5001 with two endpoints: /sentiment (text → sentiment type + confidence), /category (text → relief category + keywords). Graceful fallback chain: Python ML → EnhancedSentimentAnalyzer (Vietnamese keyword detection + emoticon matching) → SimpleSentimentAnalyzer (basic keyword counting).

8.7 Deployment & Logging

SLF4J 2.0.9 (Simple Logging Facade for Java): Abstraction layer enabling switching between logging implementations (Logback, Log4j, simple console). Configured with slf4j-simple binding for console output. Log levels: DEBUG (detailed info), INFO (key events), WARN (suspicious activity), ERROR (failures). All major operations logged: data loading, analysis execution, UI updates.

Maven Build Process: `mvn clean package` creates target/humanitarian-logistics-jar-with-dependencies.jar (fat JAR including all dependencies). Includes Maven Shade Plugin for merging JARs and avoiding conflicts.

Launch Command: `java -jar humanitarian-logistics-jar-with-dependencies.jar` starts HumanitarianLogisticsApp.main(), which orchestrates: SQLite initialization, CrawlerRegistry population, Python API startup, Swing UI creation.

8.8 Performance Characteristics

First Run: 10-15 minutes (downloading 3.6GB of ML models from Hugging Face). Subsequent runs: 30 seconds startup (models cached in ~/.cache/huggingface/hub).

Sentiment Analysis: 50-200ms per single text (depends on text length), 1-2 seconds for batch of 31 posts (parallel processing via Python). Database queries: <100ms (SQLite with indexed postId, commentId). UI chart generation: <500ms for 100 posts.

Memory Usage: ~500MB steady state (Java heap 256MB + Python models 200MB + data structures 44MB). Scales linearly with post count for 100s-1000s posts. Beyond 10,000 posts, consider pagination or PostgreSQL.

Network Usage: Initial YouTube crawl ~5-10 posts: 2-5MB (HTML downloads). Python API calls: ~5KB per request. No continuous polling; event-driven architecture via Observer pattern.

8.9 Scalability and Future Evolution

Database Scaling: Replace SQLite with PostgreSQL/MySQL for concurrent multi-user access, ACID transactions, and backup/replication. No application code changes required (DataPersistenceManager abstraction).

ML Scaling: Deploy Python backend with Gunicorn (multi-worker process server) or uWSGI for concurrent requests. Redis caching layer for sentiment results (avoid re-analyzing identical texts). Larger models (xlm-roberta-xxl) for higher accuracy if latency permits.

Frontend Scaling: Current single-user desktop Swing app. Future: web frontend (React/Vue.js) consuming REST API from Java backend, enabling multi-user concurrent analysis without modifying core logic (MVC separation).

Crawler Scaling: Current: YouTubeCrawler (YouTube watch pages). Future: FacebookCrawler, TwitterCrawler, RedditCrawler registered via CrawlerRegistry (Factory/Registry patterns enable adding without modifying existing code). Rate limiting via exponential backoff in HTTP retry logic.

8.10 Technology Stack Summary

Category	Technology	Version	Purpose
Languages	Java / Python	11+ / 3.12.7	Desktop app / ML backend
Build	Maven	3.9.11	Compilation, packaging, dependency management
Desktop UI	Swing	Built-in	GUI with zero external dependencies
Charting	JFreeChart	1.5.3	Bar, pie, time-series charts with interactivity
Database	SQLite + JDBC	3.44.0.0	Embedded persistence, two DB instances
HTTP Client	Java HttpClient / OkHttp3	Built-in / 4.11.0	YouTube crawling, API calls

Category	Technology	Version	Purpose
JSON Processing	Gson / org.json	2.10.1 / 20231013	Complex / lightweight serialization
Web Framework	Flask	2.3.0+	Python REST API for ML services
NLP Models	Transformers / PyTorch	4.30.0+ / 2.0.0+	Sentiment (xlm-roberta) / Category (BART) models
Logging	SLF4J + slf4j-simple	2.0.9	Abstraction layer with console output
Testing	JUnit	4.13.2	Unit test framework

Architecture Decisions: Lightweight HTTP-based data collection (no Selenium) for efficiency and reduced resource footprint. Microservices pattern: Java desktop app communicates with Python ML backend via REST (loose coupling, language independence). Abstraction layers (DataPersistenceManager, CrawlerRegistry) enable technology switching without breaking client code. Observer pattern ensures responsive UI without polling.

Production Readiness: Mature, battle-tested libraries (Swing since 1996, Flask since 2010, SQLite since 2000). Try-with-resources ensures resource cleanup. Prepared statements prevent SQL injection. Error handling with graceful fallbacks (Python unavailable → use Enhanced/Simple analyzers). Future evolution supported via abstraction patterns without major refactoring.