Financial Anomaly Interpretability Using LLMs

F.A.I.L

Project Report

Course: MA5750 - Object Oriented Programming

Team Members:

Aritra Dasgupta	(MA25M005)
Ashwini	(MA25M006)
Devharish N	(MA25M010)
Rumaiz Ibrahim K	(MA25M023)

Contents

Chapter 1

Project Overview

1.1 Description

A sophisticated quantitative analysis framework that demonstrates advanced Object-Oriented Programming principles, SOLID principles, and design patterns for comprehensive market analysis. The project implements a complete pipeline for detecting financial anomalies, retrieving relevant news, generating embeddings, analyzing similarities, and generating AI-powered explanations.

1.2 Key Features

- Comprehensive market data analysis
- Multiple anomaly detection strategies
- News retrieval from various sources
- Embedding generation for semantic analysis
- Similarity analysis using advanced algorithms
- AI-powered event explanations
- Progress tracking and monitoring
- App which can be hosted on Streamlit

1.3 Project Structure

```
F.A.I.L_OOPS/
├─ core/ (Core framework)
│  ├─ base.py
│  ├─ interfaces.py
│  └─ exceptions.py
├─ processors/ (Data processing)
│  ├─ data_loader.py
│  ├─ anomaly_detector.py
│  ├─ news_retriever.py
│  ├─ embedding_generator.py
│  ├─ similarity_analyzer.py
│  └─ ai_explainer.py
├─ pipeline/ (Pipeline orchestration)
│  ├─ quantitative_pipeline.py
│  ├─ progress_observer.py
│  └─ pipeline_factory.py
└─ ui/ (User interface)
   ├─ streamlit_app.py
   └─ ui_components.py
```

Chapter 2

Core OOP Concepts Implementation

2.1 Encapsulation

2.1.1 Definition

Bundling data and methods together within a class, hiding internal implementation details and providing controlled access through public methods.

2.1.2 Implementation in Project

Example 1: BaseProcessor Class

The BaseProcessor class demonstrates encapsulation by:

- **Private attributes:** Using underscore prefix (_observers) to indicate internal implementation
- **Public methods:** Providing controlled access through public methods like add_observer(), remove_observer()
- **Protected attributes:** Using protected access for logger attributes

```python
class BaseProcessor(ABC):
    def __init__(self, name: str):
        self.name = name  # Public attribute
        self._observers: List[IProgressObserver] = []  # Private
        self.logger = logging.getLogger(f"{__name__}.{name}")

    def add_observer(self, observer: IProgressObserver) -> None:
        """Public method for controlled access"""
        self._observers.append(observer)

    def _notify_progress(self, step: str,
                        progress: float, message: str) -> None:
        """Private method - internal implementation"""
        for observer in self._observers:
            observer.update_progress(step, progress, message)
```

2.1.3 Benefits

- Prevents external code from directly accessing internal state
- Allows for controlled access through public methods
- Makes the code more maintainable and less error-prone
- Enables future changes to internal implementation without affecting clients

2.2 Inheritance

2.2.1 Definition

Creating new classes based on existing classes, inheriting their properties and methods. Promotes code reuse and establishes ”is-a” relationships.

2.2.2 Example: Processor Inheritance Hierarchy

```python
# Base class
class BaseProcessor(ABC):
    def __init__(self, name: str):
        self.name = name
        self.logger = logging.getLogger(f"{__name__}.{name}")

    def log_info(self, message: str) -> None:
        self.logger.info(f"[{self.name}] {message}")

# Derived class
class DataLoader(BaseProcessor):
    def __init__(self, strategy: DataLoadStrategy):
        super().__init__("DataLoader")  # Call parent
        self.strategy = strategy

    def load_market_data(self, ticker: str, benchmark: str,
                        start_date: str, end_date: str):
        self.log_info("Loading market data")  # Inherited
        return self.strategy.load_data(ticker, benchmark,
                                      start_date, end_date)
```

2.2.3 Benefits

- **Code reuse:** Common functionality is defined once in base classes
- **Consistency:** Ensures all derived classes follow the same structure
- **Maintainability:** Changes to base classes propagate to all derived classes
- **Type safety:** Type checking ensures proper inheritance relationships

2.3 Polymorphism

2.3.1 Definition

The ability of objects of different types to be treated as instances of the same type through a common interface. Enables runtime method resolution and flexible, extensible code.

2.3.2 Example: Progress Observer Polymorphism

```python
# Common interface
class IProgressObserver(ABC):
    @abstractmethod
    def update_progress(self, step: str,
                        progress: float, message: str) -> None:
        pass

# Different implementations
class ConsoleProgressObserver(IProgressObserver):
    def update_progress(self, step: str,
                        progress: float, message: str) -> None:
        print(f"[{step}] {progress}% - {message}")

class StreamlitProgressObserver(IProgressObserver):
    def update_progress(self, step: str,
                        progress: float, message: str) -> None:
        if self.progress_bar:
            self.progress_bar.progress(progress / 100.0)

# Polymorphic usage
observers = [ConsoleProgressObserver(),
             StreamlitProgressObserver()]
for observer in observers:  # Same interface, different behavior
    observer.update_progress("analysis", 50.0, "Processing")
```

2.3.3 Benefits

- **Flexibility:** Easy to swap implementations without changing client code
- **Extensibility:** New implementations can be added without modifying existing code
- **Runtime behavior:** Behavior is determined at runtime, not compile time
- **Code reuse:** Client code can work with multiple implementations

2.4 Abstraction

2.4.1 Definition

Hiding complex implementation details and showing only essential features. Provides clear contracts and reduces cognitive load.

2.4.2 Example: Abstract Base Classes

```python
# Abstract base class - defines contract without implementation
class AnomalyDetectionStrategy(ABC):
    @abstractmethod
    def detect(self, data: pd.DataFrame,
               **kwargs) -> pd.DataFrame:
        """Detect anomalies using the specific strategy"""
        pass

    @abstractmethod
    def get_parameters(self) -> Dict[str, Any]:
        """Get the current parameters"""
        pass

# Concrete implementation
class ZScoreStrategy(AnomalyDetectionStrategy):
    def detect(self, data: pd.DataFrame,
               **kwargs) -> pd.DataFrame:
        # Complex Z-score calculation logic
        events = []
        for idx, row in data.iterrows():
            if abs(row['Z_score']) > self.z_threshold:
                events.append({
                    'Date': idx,
                    'Event_Type': 'Positive Outlier'
                        if row['Z_score'] > 0
                        else 'Negative Outlier',
                })
        return pd.DataFrame(events)
```

2.4.3 Benefits

- **Simplifies complex systems:** Clients only see what they need
- **Reduces cognitive load:** Users don't need to understand implementation details
- **Provides clear contracts:** Interfaces define what methods must be implemented
- **Enables multiple implementations:** Different classes can implement the same interface

Chapter 3

SOLID Principles Implementation

3.1 Single Responsibility Principle (SRP)

3.1.1 Definition

A class should have only one reason to change. Each class should have a single, well-defined responsibility.

3.1.2 Implementation

```python
# Single responsibility: Data loading only
class DataLoader(BaseProcessor):
    def load_market_data(self, ticker: str, benchmark: str,
                         start_date: str, end_date: str):
        return self.strategy.load_data(ticker, benchmark,
                                       start_date, end_date)

# Single responsibility: Data persistence only
class BaseDataRepository(IDataRepository):
    def save_events(self, events: pd.DataFrame,
                    filename: str) -> None:
        filepath = f"{self.base_path}/{filename}"
        events.to_parquet(filepath)

    def load_events(self, filename: str) -> pd.DataFrame:
        filepath = f"{self.base_path}/{filename}"
        return pd.read_parquet(filepath)
```

3.1.3 Focused Classes

Each processor class has a single, clear responsibility:

- **DataLoader:** Only responsible for loading market data
- **AnomalyDetector:** Only responsible for detecting anomalies
- **NewsRetriever:** Only responsible for retrieving news
- **EmbeddingGenerator:** Only responsible for generating embeddings

- **SimilarityAnalyzer:** Only responsible for similarity analysis
- **AIExplainer:** Only responsible for generating AI explanations

3.2 Open/Closed Principle (OCP)

3.2.1 Definition

Software entities should be open for extension but closed for modification. New functionality should be added through extension, not by modifying existing code.

3.2.2 Example: Strategy Pattern for Extension

```python
# Base class - closed for modification
class AnomalyDetectionStrategy(ABC):
    @abstractmethod
    def detect(self, data: pd.DataFrame,
               **kwargs) -> pd.DataFrame:
        pass

# Existing implementation - no modification needed
class ZScoreStrategy(AnomalyDetectionStrategy):
    def detect(self, data: pd.DataFrame,
               **kwargs) -> pd.DataFrame:
        # Z-score implementation
        pass

# New implementation - extension without modification
class NewAnomalyStrategy(AnomalyDetectionStrategy):
    def detect(self, data: pd.DataFrame,
               **kwargs) -> pd.DataFrame:
        # New anomaly detection algorithm
        pass

# Client code works with all strategies
detector = AnomalyDetector(NewAnomalyStrategy())
```

3.3 Liskov Substitution Principle (LSP)

3.3.1 Definition

Objects of a superclass should be replaceable with objects of a subclass without breaking functionality. Derived classes must honor the contracts of their base classes.

3.3.2 Example: Strategy Substitution

```python
# Base class contract
class AnomalyDetectionStrategy(ABC):
    @abstractmethod
    def detect(self, data: pd.DataFrame,
               **kwargs) -> pd.DataFrame:
```

```python
        pass

# Subclass implementations honor the contract
class ZScoreStrategy(AnomalyDetectionStrategy):
    def detect(self, data: pd.DataFrame,
              **kwargs) -> pd.DataFrame:
        return events_dataframe

class IsolationForestStrategy(AnomalyDetectionStrategy):
    def detect(self, data: pd.DataFrame,
              **kwargs) -> pd.DataFrame:
        return events_dataframe

# LSP compliance - both can be used interchangeably
detector1 = AnomalyDetector(ZScoreStrategy())
detector2 = AnomalyDetector(IsolationForestStrategy())

# Both work the same way
events1 = detector1.detect(data)
events2 = detector2.detect(data)
```

3.4 Interface Segregation Principle (ISP)

3.4.1 Definition

Clients should not be forced to depend on interfaces they don't use. Create specific, focused interfaces rather than large, general-purpose ones.

3.4.2 Example: Segregated Interfaces

```python
# Good: Segregated interfaces (used in project)
class IDataLoader(ABC):
    @abstractmethod
    def load_data(self): pass

class IDataRepository(ABC):
    @abstractmethod
    def save_data(self): pass

class IProgressObserver(ABC):
    @abstractmethod
    def update_progress(self, step: str,
                       progress: float, message: str) -> None:
        pass

class IAnomalyDetector(ABC):
    @abstractmethod
    def detect(self, data: pd.DataFrame) -> pd.DataFrame:
        pass
```

3.5 Dependency Inversion Principle (DIP)

3.5.1 Definition

High-level modules should not depend on low-level modules. Both should depend on abstractions. Depend on abstractions, not concretions.

3.5.2 Example: Dependency Injection

```python
# Good: Depends on abstractions (used in project)
class QuantitativeAnalysisPipeline(BaseAnalysisPipeline):
    def __init__(self, config: Dict[str, Any]):
        # Dependencies created through factories (abstractions)
        self.data_loader = DataLoader(YFinanceDataStrategy())
        self.anomaly_detector = AnomalyDetector(ZScoreStrategy())
        # Or through factory methods
        self.news_retriever = NewsRetrieverFactory.create_retriever(
            'finnhub', api_key)
```

Chapter 4

Design Patterns Implementation

4.1 Strategy Pattern

4.1.1 Definition

Define a family of algorithms, encapsulate each one, and make them interchangeable. Allows runtime selection of algorithms.

4.1.2 Implementation Areas

The project extensively uses Strategy pattern for:

- Data loading (YFinanceDataStrategy, AlphaVantageDataStrategy)
- Anomaly detection (ZScoreStrategy, IsolationForestStrategy)
- News retrieval (FinnhubNewsStrategy, YahooNewsStrategy)
- Embedding generation (SentenceTransformerStrategy, OpenAIEmbeddingStrategy)
- Similarity analysis (FAISSStrategy, CosineSimilarityStrategy)
- AI explanations (GroqExplainerStrategy, OpenAIExplainerStrategy)

4.1.3 Example Code

```python
# Strategy interface
class AnomalyDetectionStrategy(ABC):
    @abstractmethod
    def detect(self, data: pd.DataFrame,
               **kwargs) -> pd.DataFrame:
        pass

# Concrete strategies
class ZScoreStrategy(AnomalyDetectionStrategy):
    def detect(self, data: pd.DataFrame,
               **kwargs) -> pd.DataFrame:
        events = []
        for idx, row in data.iterrows():
            if abs(row['Z_score']) > self.z_threshold:
                events.append({
                    'Date': idx,
                    'Event_Type': 'Positive Outlier'
                        if row['Z_score'] > 0
                        else 'Negative Outlier',
                })
        return pd.DataFrame(events)

# Context class
class AnomalyDetector:
    def __init__(self, strategy: AnomalyDetectionStrategy):
        self.strategy = strategy

    def detect(self, data: pd.DataFrame,
              **kwargs) -> pd.DataFrame:
        return self.strategy.detect(data, **kwargs)
```

4.2 Factory Pattern

4.2.1 Definition

Create objects without specifying their exact class. Encapsulates object creation logic.

4.2.2 Example: News Retriever Factory

```
class NewsRetrieverFactory:
    @staticmethod
    def create_retriever(service: str,
                         api_key: str = None) -> NewsRetriever:
        if service.lower() == 'finnhub':
            strategy = FinnhubNewsStrategy(api_key)
        elif service.lower() == 'yahoo':
            strategy = YahooNewsStrategy()
        else:
            raise ValueError(f"Unknown service: {service}")

        return NewsRetriever(strategy)

# Usage
retriever = NewsRetrieverFactory.create_retriever(
    'finnhub', api_key='your_key')
```

4.3 Observer Pattern

4.3.1 Definition

Define a one-to-many dependency between objects so that when one object changes state, all dependents are notified.

4.3.2 Example: Progress Observer

```python
# Observer interface
class IProgressObserver(ABC):
    @abstractmethod
    def update_progress(self, step: str, progress: float,
                        message: str) -> None:
        pass

# Subject (Observable)
class BaseProcessor(ABC):
    def __init__(self, name: str):
        self._observers: List[IProgressObserver] = []

    def add_observer(self,
                     observer: IProgressObserver) -> None:
        self._observers.append(observer)

    def notify_progress(self, step: str, progress: float,
                        message: str) -> None:
        for observer in self._observers:
            observer.update_progress(step, progress, message)

# Usage
processor = DataLoader(YFinanceDataStrategy())
processor.add_observer(ConsoleProgressObserver())
processor.notify_progress("data_loading", 50.0,
                          "Loading market data...")
```

4.4 Template Method Pattern

4.4.1 Definition

Define the skeleton of an algorithm in a base class, letting subclasses override specific steps.

4.4.2 Example: Pipeline Template Method

```python
# Abstract base class with template method
class BaseAnalysisPipeline(ABC):
    def run(self, **kwargs) -> Dict[str, Any]:
        """Template method - defines algorithm structure"""
        self.logger.info(f"Starting {self.name} pipeline")
        self.notify_progress("start", 0.0,
                             f"Starting {self.name}")

        # Template method steps
        self._preprocess(**kwargs)
        self._analyze(**kwargs)
        self._postprocess(**kwargs)

        self.notify_progress("complete", 100.0,
                             f"{self.name} completed")
        return self._results

    @abstractmethod
    def _preprocess(self, **kwargs) -> None:
        pass

    @abstractmethod
    def _analyze(self, **kwargs) -> None:
        pass

    @abstractmethod
    def _postprocess(self, **kwargs) -> None:
        pass
```

4.5 Builder Pattern

4.5.1 Definition

Construct complex objects step by step. Provides flexibility in object construction and improves readability.

4.5.2 Example: Pipeline Builder

```python
class PipelineBuilder:
    def __init__(self):
        self.config = {}
        self.observers = []

    def with_ticker(self, ticker: str) -> 'PipelineBuilder':
        self.config['ticker'] = ticker
        return self

    def with_benchmark(self,
                       benchmark: str) -> 'PipelineBuilder':
        self.config['benchmark'] = benchmark
        return self

    def build(self) -> QuantitativeAnalysisPipeline:
        return PipelineFactory.create_quantitative_pipeline(
            self.config, self.observers)

# Usage - fluent interface
pipeline = (PipelineBuilder()
    .with_ticker('TSLA')
    .with_benchmark('SPY')
    .with_anomaly_detection(z_threshold=3.0)
    .build())
```

4.6 Repository Pattern

4.6.1 Definition

Encapsulate data access logic and provide a uniform interface. Separates data access logic from business logic.

4.6.2 Example: Data Repository

```python
# Repository interface
class IDataRepository(ABC):
    @abstractmethod
    def save_events(self, events: pd.DataFrame,
                    filename: str) -> None:
        pass

    @abstractmethod
    def load_events(self, filename: str) -> pd.DataFrame:
        pass

# Concrete implementation
class BaseDataRepository(IDataRepository):
    def __init__(self, base_path: str = "results"):
        self.base_path = base_path
        os.makedirs(base_path, exist_ok=True)

    def save_events(self, events: pd.DataFrame,
                    filename: str) -> None:
        filepath = os.path.join(self.base_path, filename)
        events.to_parquet(filepath)
```

Chapter 5

Architecture Overview

5.1 Layered Architecture

The project follows a layered architecture with clear separation of concerns:

Layer 1: Presentation Layer UI Components (Streamlit app, console interface), User interaction handling, Progress display and feedback

Layer 2: Business Logic Layer Pipeline orchestration (Template Method), Component coordination, Workflow management

Layer 3: Processing Layer Data processors (Strategy Pattern), Algorithm implementations, Business logic execution

Layer 4: Data Access Layer Data repositories (Repository Pattern), Data persistence, Data retrieval

Layer 5: Infrastructure Layer External API integrations, File system operations, Logging and error handling

5.2 UML Diagrams

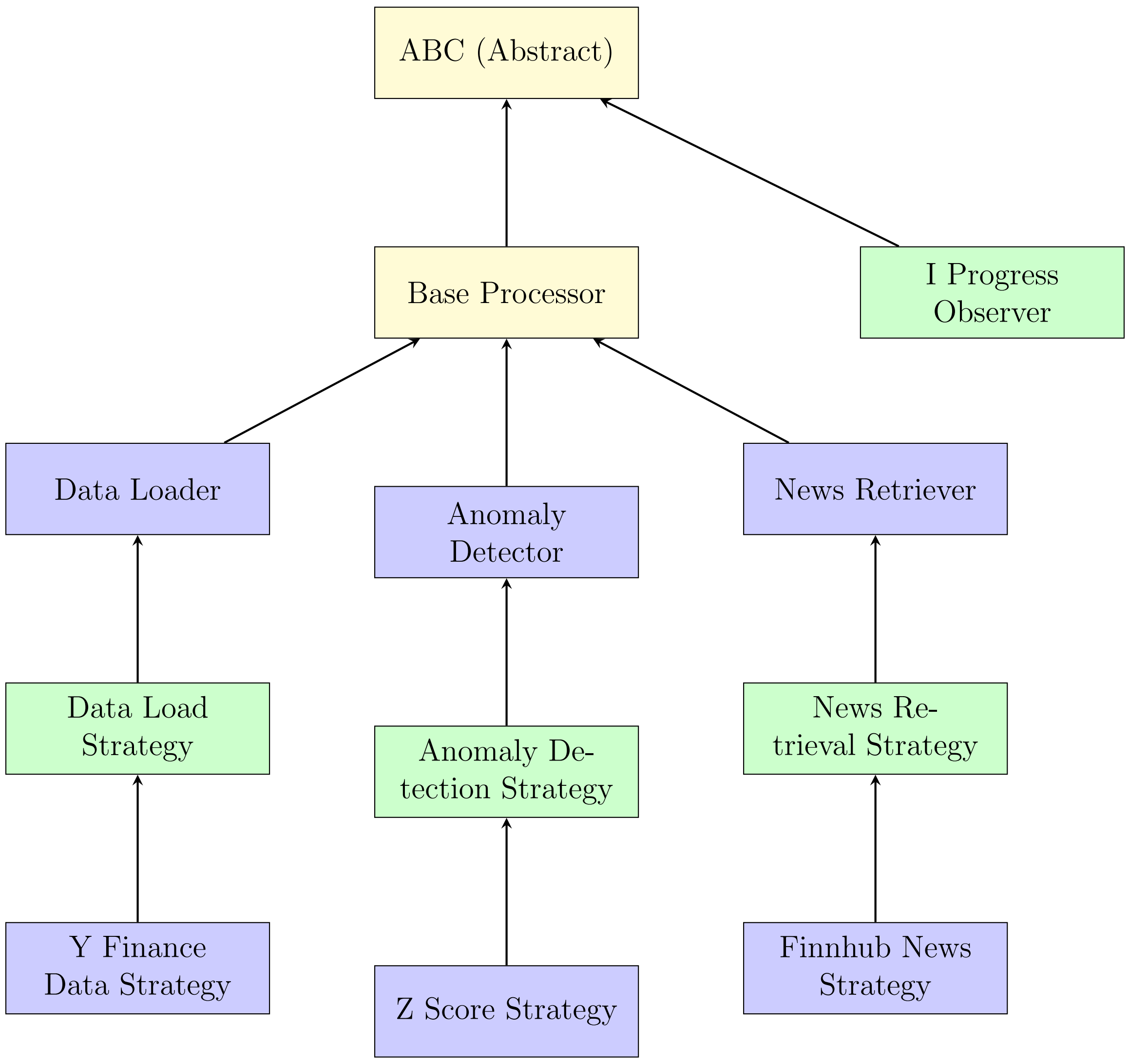

Figure 5.1: Core Processor Hierarchy - Inheritance and Strategy Pattern

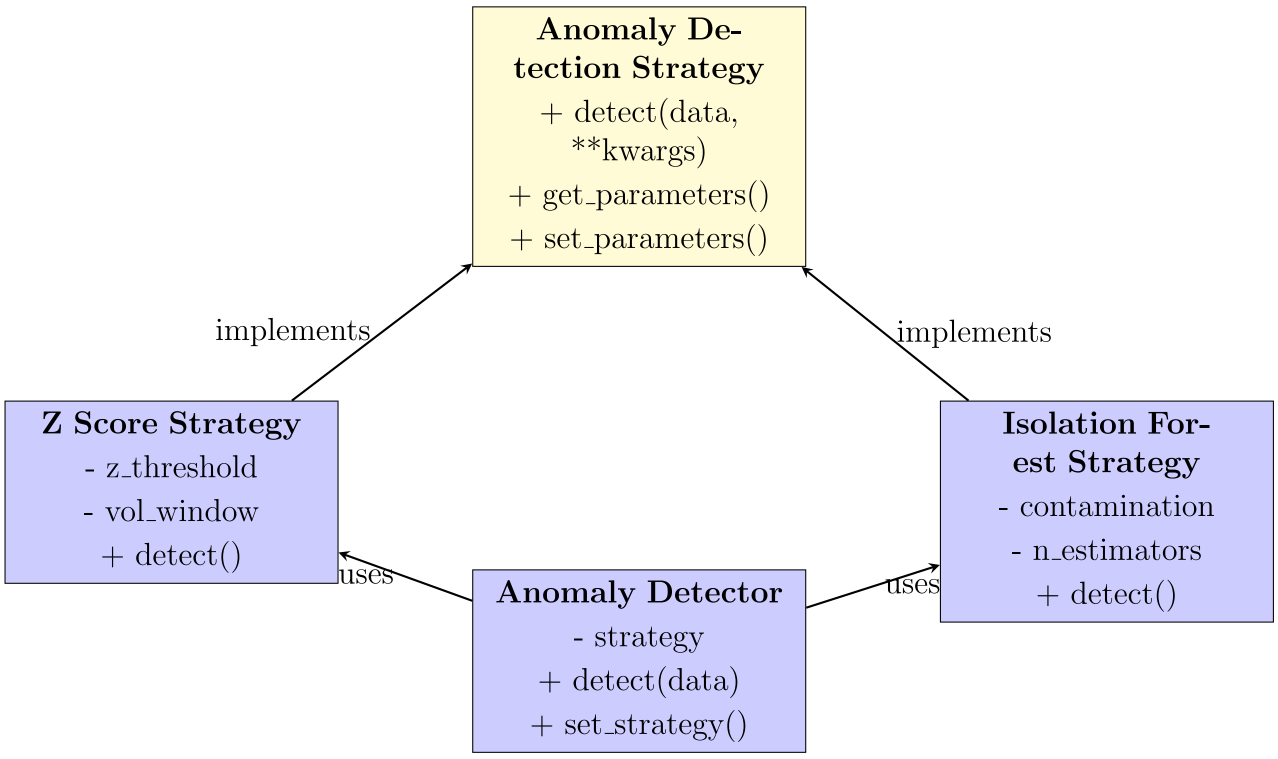

Figure 5.2: Strategy Pattern for Anomaly Detection

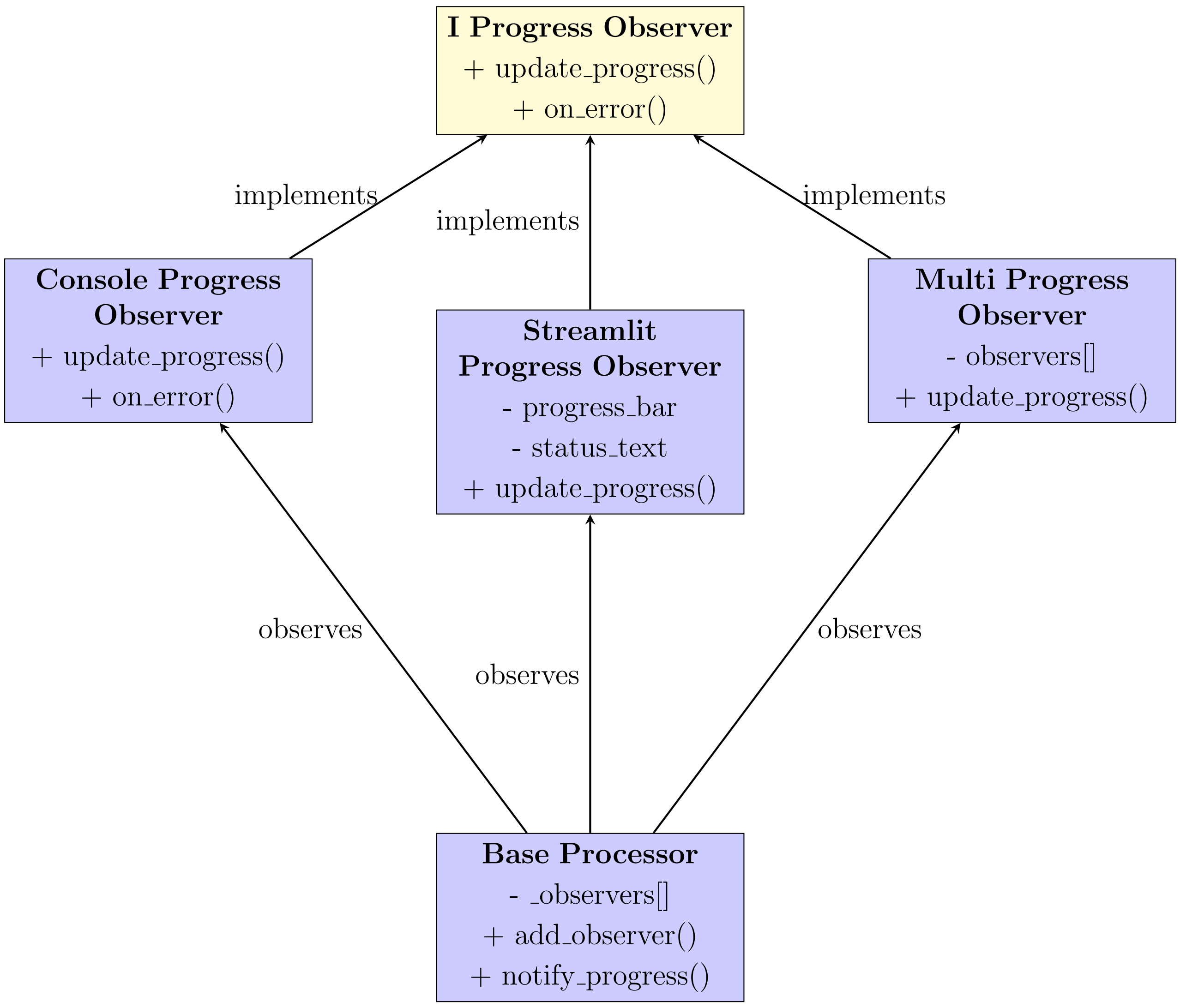

Figure 5.3: Observer Pattern for Progress Tracking

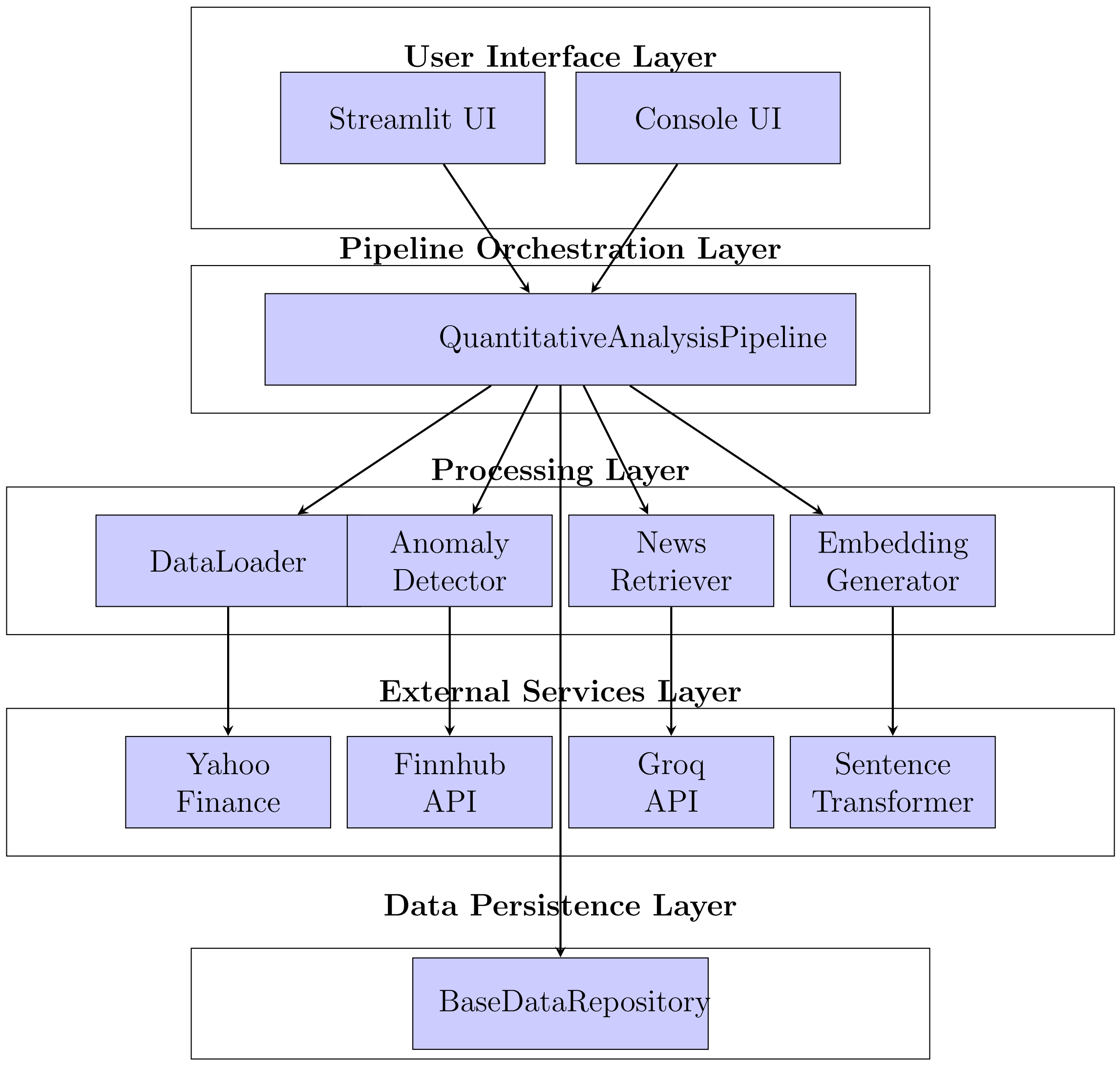

Figure 5.4: System Component Architecture

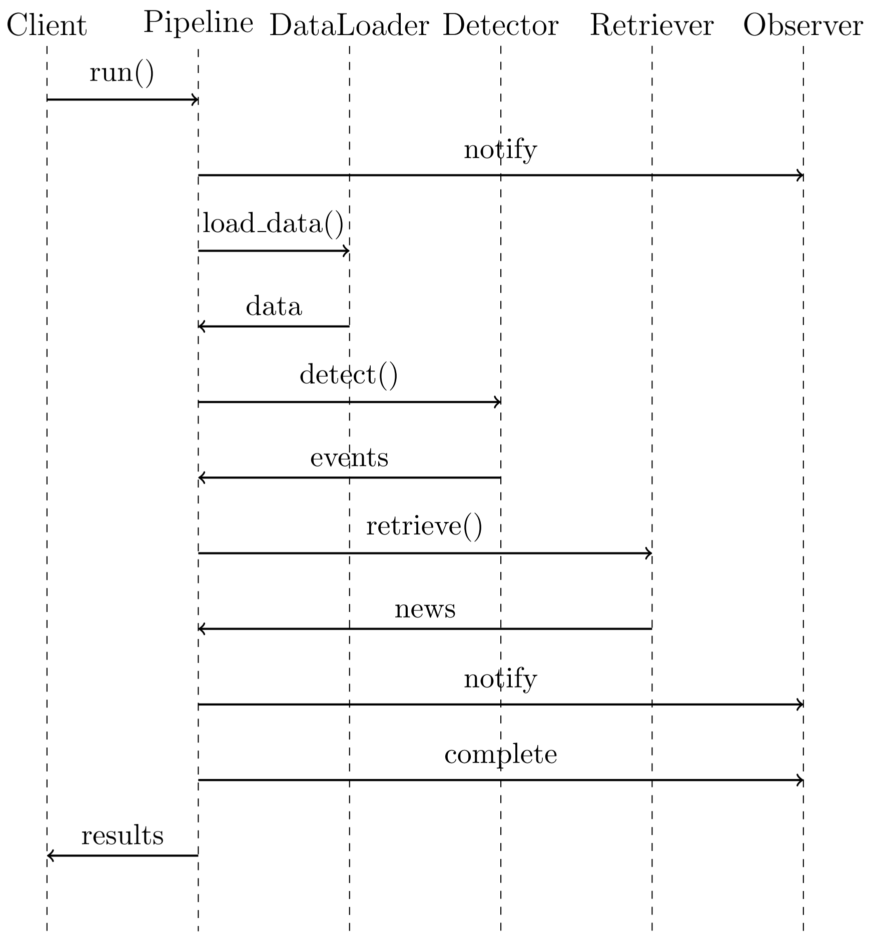

Figure 5.5: Sequence Diagram - Pipeline Execution Flow

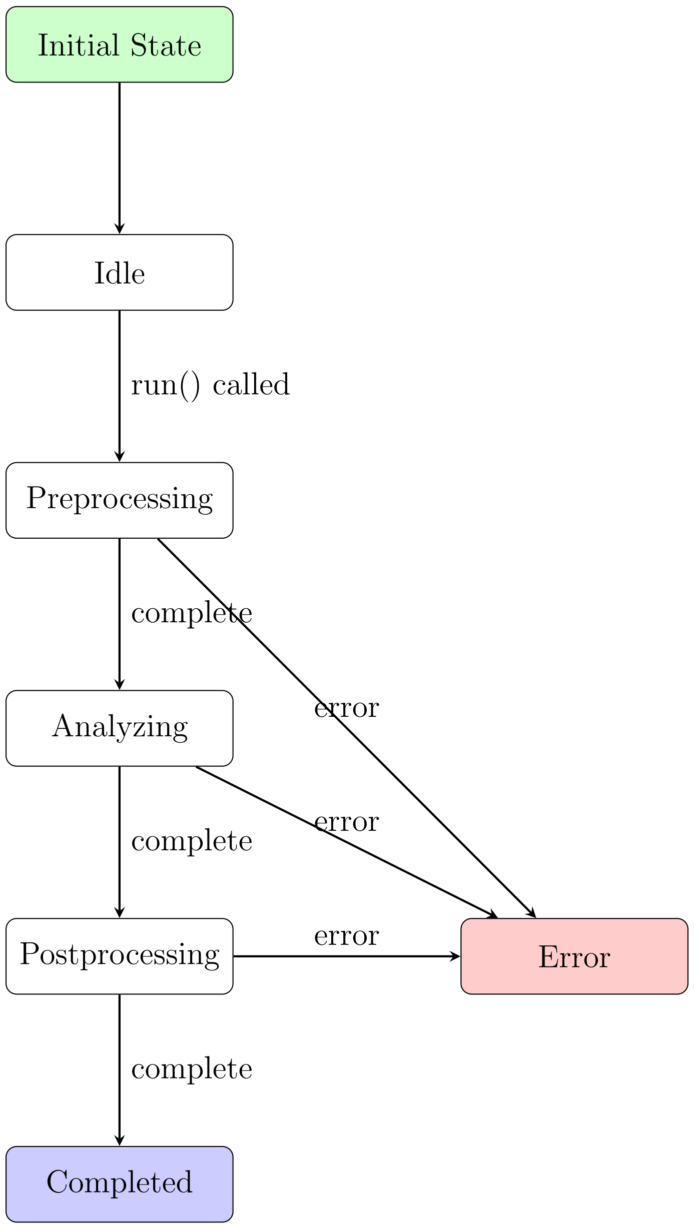

Figure 5.6: State Diagram - Pipeline Execution States

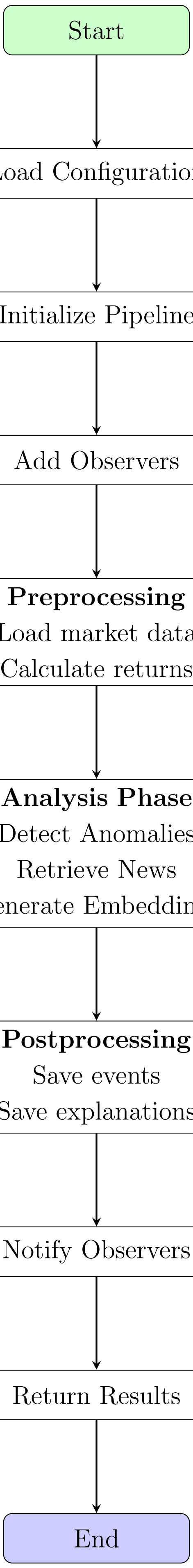

Figure 5.7: Activity Diagram - Complete Analysis Workflow

Chapter 6

Code Examples and Demonstrations

6.1 Complete Pipeline Execution

```python
from pipeline.pipeline_factory import PipelineBuilder
from pipeline.progress_observer import ConsoleProgressObserver

# Using Builder pattern
pipeline = (PipelineBuilder()
    .with_ticker('TSLA')
    .with_benchmark('SPY')
    .with_date_range('2024-01-01', '2024-12-31')
    .with_anomaly_detection(z_threshold=3.0, vol_window=15)
    .with_news_service('finnhub', api_key='your_key')
    .with_ai_service('groq', api_key='your_key')
    .with_console_observer()
    .build())

# Run analysis
results = pipeline.run()
```

6.2 Custom Strategy Implementation

```python
from processors.anomaly_detector import (
    AnomalyDetectionStrategy, AnomalyDetector
)

class CustomAnomalyStrategy(AnomalyDetectionStrategy):
    def detect(self, data: pd.DataFrame,
               **kwargs) -> pd.DataFrame:
        # Custom anomaly detection logic
        events = []
        # ... implementation
        return pd.DataFrame(events)

    def get_parameters(self) -> Dict[str, Any]:
        return {'custom_param': 'value'}

    def set_parameters(self, **kwargs) -> None:
        # Set custom parameters
        pass

# Use custom strategy
detector = AnomalyDetector(CustomAnomalyStrategy())
events = detector.detect(data)
```

6.3 Multiple Observers

```python
from pipeline.progress_observer import (
    ConsoleProgressObserver,
    StreamlitProgressObserver,
    MultiProgressObserver
)

# Create multiple observers
console_observer = ConsoleProgressObserver()
streamlit_observer = StreamlitProgressObserver(
    progress_bar, status_text
)

# Create multi-observer
multi_observer = MultiProgressObserver([
    console_observer,
    streamlit_observer
])

# Add to pipeline
pipeline.add_observer(multi_observer)
```

Chapter 7

Benefits and Best Practices

7.1 Benefits of OOP in This Project

1. **Maintainability:** Easy to modify and extend individual components
2. **Reusability:** Components can be reused in different contexts
3. **Testability:** Each component can be tested in isolation
4. **Flexibility:** Easy to swap implementations using interfaces
5. **Scalability:** New features can be added without modifying existing code
6. **Readability:** Code structure mirrors real-world concepts
7. **Type Safety:** Type hints and interfaces ensure correct usage
8. **Documentation:** Clear interfaces serve as documentation

7.2 Best Practices Demonstrated

1. **Use Interfaces for Contracts:** Define clear contracts between components
2. **Favor Composition over Inheritance:** Use object composition for flexibility
3. **Single Responsibility:** Each class has one clear purpose
4. **Dependency Injection:** Inject dependencies rather than creating them
5. **Error Handling:** Comprehensive exception hierarchy
6. **Logging:** Proper logging throughout the application
7. **Type Hints:** Use type hints for better code documentation
8. **Documentation:** Clear documentation for all public methods
9. **Testing:** Design for testability
10. **SOLID Principles:** All SOLID principles properly applied

7.3 Design Pattern Benefits Summary

Pattern	**Benefits**
Strategy	Runtime algorithm selection, easy to add new algorithms, separates logic from client code
Factory	Encapsulates object creation, provides flexibility, reduces coupling
Observer	Loose coupling, supports multiple observers, event-driven architecture
Template Method	Defines common algorithm structure, allows customization, promotes code reuse
Builder	Step-by-step construction, fluent interface, flexible configuration
Repository	Abstracts data access, uniform interface, easy to swap storage

Table 7.1: Design Pattern Benefits Summary

Chapter 8

Conclusion

This project successfully demonstrates:

8.1 Comprehensive OOP Implementation

✓ **Encapsulation:** Data and methods bundled with controlled access

✓ **Inheritance:** Code reuse through class hierarchies

✓ **Polymorphism:** Runtime behavior selection through interfaces

✓ **Abstraction:** Complex implementation hidden behind clear contracts

8.2 SOLID Principles

✓ **Single Responsibility Principle:** Each class has one responsibility

✓ **Open/Closed Principle:** Open for extension, closed for modification

✓ **Liskov Substitution Principle:** Subclasses honor base class contracts

✓ **Interface Segregation Principle:** Focused, specific interfaces

✓ **Dependency Inversion Principle:** Depends on abstractions, not concretions

8.3 Design Patterns

✓ **Strategy Pattern:** 6+ implementations across different domains

✓ **Factory Pattern:** 4+ factories for object creation

✓ **Observer Pattern:** Comprehensive progress tracking system

✓ **Template Method Pattern:** Structured pipeline execution

✓ **Builder Pattern:** Fluent interface for configuration

✓ **Repository Pattern:** Clean data persistence layer

8.4 Professional Software Engineering

- ✓ Clean, maintainable, and extensible code
- ✓ Well-structured, layered architecture
- ✓ Industry-standard coding practices
- ✓ Comprehensive error handling and logging

The project serves as an excellent example of how Object-Oriented Programming principles and design patterns can be applied to create robust, maintainable, and extensible software systems. The implementation demonstrates deep understanding of OOP concepts and their practical application in solving real-world problems.