TED UNIVERSITY

CMPE492/SENG492 Senior Project II

SyntaxSavior Project Low-Level Design Report

09.03.2025

Team Members:

İrem BEŞİROĞLU Software Engineering

Yiğit Oğuzhan KÖKTEN Computer Engineering

Yüksel Çağlar BAYPINAR Computer Engineering

Table of Contents

1. Introduction

1.1. Object design trade-offs

The chosen design decisions, as explained in the following sections, prioritize educational impact, resource efficiency, and scalability, while avoiding over-complexity and over-reliance on automation. Alternatives were rejected due to their inability to meet the system's pedagogical goals or their impracticality given project constraints. These trade-offs ensure that SyntaxSavior remains a functional, user-friendly, and educationally effective tool for introductory programming students.

1.1.1. Manual vs. Automatic Code Monitoring

Chosen Approach: Hybrid Monitoring (Manual Trigger + Automatic Surface-Level Feedback)

- **Pros:**
    - **User Control:** Students can manually trigger in-depth analysis, fostering active engagement and reducing over-reliance on automated tools.
    - **Reduced Cognitive Load:** Automatic surface-level feedback (e.g., syntax errors, missing brackets) ensures immediate assistance without overwhelming the student.
    - **Resource Efficiency:** Delayed automatic feedback reduces continuous background processing, minimizing resource usage on lab machines.
- **Cons:**
    - **Latency in Feedback:** Automatic feedback may have a slight delay, which could frustrate students expecting instant results.
    - **Complexity in Implementation:** Balancing manual and automatic triggers requires careful state management and event handling.

Alternatives:

1. **Fully Automatic Monitoring:**
    - **Pros:** Immediate feedback on all errors, reducing the need for manual intervention.
    - **Cons:** Overwhelms students with constant notifications, risks over-reliance, and increases resource usage.
    - **Why Not Chosen:** Contradicts the goal of fostering independent problem-solving skills.

2. **Fully Manual Monitoring:**
    - **Pros:** Complete user control, minimal resource usage.
    - **Cons:** Students may miss critical errors, leading to frustration and slower progress.
    - **Why Not Chosen:** Fails to provide timely assistance for common, easily fixable errors.

1.1.2. REST API vs. WebSocket for Communication

Chosen Approach: REST API

- **Pros:**
    - **Simplicity:** REST is well-understood, easy to implement, and widely supported.
    - **Statelessness:** Each request is independent, simplifying error handling and scaling.
    - **Compatibility:** Works seamlessly with existing development environment APIs and backend frameworks.
- **Cons:**
    - **Latency:** Each request incurs overhead, which may slightly delay feedback.
    - **Polling Requirement:** For real-time updates, the plugin may need to poll the backend periodically.

Alternatives:

1. **WebSocket:**
    - **Pros:** Real-time, bidirectional communication, ideal for instant feedback.
    - **Cons:** Increased complexity in implementation and state management.
    - **Why Not Chosen:** Overkill for a hybrid monitoring system where real-time updates are not always required.
2. **gRPC:**
    - **Pros:** High performance, supports streaming, and strong typing.
    - **Cons:** Steeper learning curve and less flexibility compared to REST.
    - **Why Not Chosen:** REST provides sufficient performance and is easier to integrate with existing tools.

1.1.3. Student Error Handling Strategies

Chosen Approach: Multi-Layered Error Classification and Feedback

- **Pros:**
  - **Granular Feedback:** Errors are categorized into syntax, runtime, and logical types, providing tailored feedback for each.
  - **Contextual Guidance:** Feedback is aligned with the lab task and course progression, ensuring relevance.
  - **Educational Focus:** Explanations emphasize understanding over quick fixes (e.g., "You forgot a semicolon, which terminates a statement in Java" instead of just "Missing semicolon").
- **Cons:**
  - **Complexity in Implementation:** Requires robust parsing and analysis logic to accurately classify errors.
  - **Maintenance Overhead:** Error classification rules must be updated as the curriculum evolves.

Alternatives:

1. **Generic Error Messages:**
   - **Pros:** Easier to implement and maintain.
   - **Cons:** Provides less actionable feedback, hindering student learning.
   - **Why Not Chosen:** Fails to meet the educational goals of the system.
2. **Direct Solutions:**
   - **Pros:** Immediate access to fixes for errors, reducing student frustration.
   - **Cons:** Encourages dependency on the tool, undermining critical thinking skills.
   - **Why Not Chosen:** Contradicts the pedagogical philosophy of SyntaxSavior.

1.1.4. AI Model Training: Fine-Tuning vs. Training from Scratch

Chosen Approach: Fine-Tuning an Existing Model

- **Pros:**
  - **Cost-Effective:** Leverages pre-trained models (e.g., GPT, Codex), reducing computational and financial costs.
  - **Faster Deployment:** Fine-tuning requires less time and data compared to training from scratch.

- **State-of-the-Art Performance:** Pre-trained models already understand programming concepts, making them ideal for educational feedback.

- **Cons:**
  - **Limited Customization:** Fine-tuning may not fully capture domain-specific nuances (e.g., CMPE113 lab tasks).
  - **Dependency on External Models:** Relies on the availability and licensing of pre-trained models.

Alternatives:

1. **Training from Scratch:**
   - **Pros:** Complete control over model behavior and domain-specific optimization.
   - **Cons:** Requires massive datasets, computational resources, and time.
   - **Why Not Chosen:** Infeasible given budget and time constraints.
2. **Rule-Based Systems:**
   - **Pros:** Transparent, easy to debug, and fully customizable.
   - **Cons:** Limited flexibility and scalability, unable to handle complex or novel errors.
   - **Why Not Chosen:** Insufficient for providing nuanced, context-aware feedback.

1.1.5. Vector Database Integration

Chosen Approach: Use of Vector Database for Contextual Retrieval

- **Pros:**
  - **Efficient Similarity Search:** Enables fast retrieval of relevant course materials and examples based on code context.
  - **Scalability:** Handles large datasets (e.g., course materials, FAQs) with low latency.
  - **Dynamic Updates:** Supports real-time updates to the dataset as new materials are added.
- **Cons:**
  - **Complexity in Setup:** Requires expertise in vector embeddings and database management.
  - **Resource Usage:** May increase backend resource requirements.

Alternatives:

1. **Relational Database:**
   - **Pros:** Simpler to implement and query.
   - **Cons:** Inefficient for similarity-based searches, limiting contextual relevance.
   - **Why Not Chosen:** Fails to meet the need for dynamic, context-aware retrieval.
2. **NoSQL Database:**
   - **Pros:** Flexible schema, suitable for unstructured data.
   - **Cons:** Lacks native support for vector-based similarity searches.
   - **Why Not Chosen:** Does not align with the requirement for semantic search capabilities.

1.2. Interface documentation guidelines

This section provides **guidelines** and **standards** for documenting the API contracts, communication protocols, and error response formats for the SyntaxSavior system. These guidelines ensure consistency, clarity, and maintainability in the design and implementation of the interfaces.

1.2.1. API Contract Guidelines

The API contracts define how the **IDE Plugin** and **Backend Server** communicate. Below are the standards we will follow:

1. **Base URL Structure:**
   - Using a consistent base URL for all API endpoints.
   - Example: https://api.syntaxsavior.com/v1
   - Including versioning (e.g., /v1) to allow for future updates without breaking existing integrations.
2. **Endpoint Naming Conventions:**
   - Using **lowercase** and **hyphen-separated** names for endpoints.
   - Example: /analyze-code, /surface-feedback
   - Avoiding verbs in endpoint names (e.g., use /analyze instead of /analyzeCode).
3. **HTTP Methods:**
   - Using appropriate HTTP methods for each endpoint:
     - GET: Retrieve data (e.g., fetch course materials).
     - POST: Submit data (e.g., send code for analysis).
     - PUT: Update data (e.g., update user preferences).

        - DELETE: Remove data (e.g., delete a session).

4. **Request and Response Formats:**
    - Using **JSON** for all request and response bodies.
    - Including clear and descriptive field names.
    - Example request body:

```
{
  "code": "public class Main { ... }",
  "task_id": "lab-1",
  "course_progress": "Loops"
}
```

Figure 1:Request Body Example

    - Example response body:

```
{
  "status": "success",
  "errors": [
    {
      "type": "syntax",
      "message": "Missing semicolon at line 5",
      "line": 5,
      "hint": "Add a semicolon (;) at the end of the statement."
    }
  ],
  "concept_explanation": "In Java, statements must end with a semicolon."
}
```

Figure 2: Response Body Example

5. **Error Handling:**
    - Using consistent error response formats for all endpoints.
    - Including a status field to indicate success or failure.
    - Example error response:

```
{
  "status": "error",
  "message": "Invalid request format: Missing 'code' field."
}
```

Figure 3: Error Response Example

6. **Authentication and Authorization:**

- Using **Bearer Tokens** for authentication.
- Including an Authorization header in all requests.

1.2.2. Communication Protocol Guidelines

The communication between the IDE plugin and backend server will follow these standards:

1. **RESTful Principles:**
   - Using RESTful design principles, including statelessness and resource-based endpoints.
   - Avoiding session state on the server; instead, include all necessary information in each request.
2. **Request Throttling:**
   - Implementing rate limiting to prevent abuse (e.g., max 10 requests per second per user).
   - Return a 429 Too Many Requests response if the limit is exceeded.
3. **Data Validation:**
   - Validating all incoming requests on the backend to ensure data integrity.
   - Returning descriptive error messages for invalid requests.
4. **Versioning:**
   - Including versioning in the API to allow for future updates.
   - Example: /v1/analyze

1.2.3. Error Response Guidelines

Error responses will be consistent and informative to help developers debug issues.

1. **Standard Error Fields:**
   - Including the following fields in all error responses:

```
{
  "status": "error",
  "message": "Descriptive error message",
  "code": "ERROR_CODE", // Optional: Internal error code
  "details": "Additional details or context" // Optional
}
```

Figure 4 :Error Response With Further Details

2. **Common Error Codes:**
    - Defining a set of standard error codes for common issues:
        - 400: Bad Request (e.g., missing or invalid fields).
        - 401: Unauthorized (e.g., missing or invalid token).
        - 404: Not Found (e.g., resource does not exist).
        - 429: Too Many Requests (e.g., rate limit exceeded).
        - 500: Internal Server Error (e.g., unexpected server error).
3. **User-Friendly Messages:**
    - Ensuring error messages are clear and actionable for end-users or tutors in class.

1.2.4. Documentation Format

The API documentation should follow a clear and consistent format. Use tools like **Swagger** or **Postman** to generate interactive documentation.

1. **Endpoint Documentation:**
    - For each endpoint, include:
        - **Description**: Purpose of the endpoint.
        - **HTTP Method**: GET, POST, etc.
        - **URL**: Full endpoint URL.
        - **Request Body**: Example request with all fields explained.
        - **Response Body**: Example response with all fields explained.
        - **Error Responses**: List of possible error responses.
2. **Example Documentation:**

````
### Analyze Code
**Description**: Analyzes a student's code and provides feedback.
**Method**: `POST`
**URL**: `/v1/analyze`
**Request Body**:
```json
{
  "code": "public class Main { ... }",
  "task_id": "lab-1",
  "course_progress": "Loops"
}
````

Figure 5: Example Markdown for Documentation
````

Do keep in mind this is simply a template, and not descriptive of the ideal documentation or functionality.

1.3. Engineering standards (e.g., UML and IEEE)

This section outlines the **engineering standards** that will be adhered to during the design, development, and maintenance of the SyntaxSavior system. These standards ensure that the project is **well-documented**, **maintainable**, and **scalable**, while following industry best practices.

1.3.1. Adherence to IEEE 1016-2009 (System Design Documentation)

The IEEE 1016-2009 standard provides guidelines for creating **system design documentation**. SyntaxSavior will follow these guidelines to ensure clarity, consistency, and completeness in its documentation.

1. **Document Structure:**
    - **Introduction**: Overview of the system, its purpose, and scope.
    - **System Architecture**: High-level description of subsystems and their interactions.
    - **Detailed Design**: Low-level design of components, including class diagrams, sequence diagrams, and interface specifications.
    - **Data Management**: Description of data storage, retrieval, and processing mechanisms.
    - **Testing and Validation**: Outline of testing strategies and validation procedures.
2. **Key Deliverables:**
    - **System Design Document (SDD)**: Comprehensive document covering all aspects of the system design.
    - **Interface Specification Document (ISD)**: Detailed description of APIs, communication protocols, and error handling.
    - **User Manual**: Guide for end-users (students, instructors) on how to use the system.
3. **UML 2.5 Compliance:**
    - Use **Unified Modeling Language (UML) 2.5** for creating diagrams, including:
        - **Class Diagrams**: To represent the structure of the system.
        - **Sequence Diagrams**: To illustrate interactions between components.
        - **Activity Diagrams**: To depict workflows and processes.

    - Tools like **PlantUML** or **Mermaid** will be used to generate and maintain these diagrams.

1.3.2. Code Quality Standards

To ensure high-quality code, SyntaxSavior will adhere to the following standards:

1. **SOLID Principles:**
    - **Single Responsibility Principle (SRP)**: Each class or module should have only one reason to change.
        - Example: The SyntaxAnalyzer class is responsible only for detecting syntax errors.
    - **Open/Closed Principle (OCP)**: Classes should be open for extension but closed for modification.
        - Example: Adding new error types should not require changes to the Error class.
    - **Liskov Substitution Principle (LSP)**: Subclasses should be substitutable for their base classes.
        - Example: Any subclass of CodeMonitor should work seamlessly in place of the base class.
    - **Interface Segregation Principle (ISP)**: Clients should not be forced to depend on interfaces they do not use.
        - Example: Separate interfaces for SyntaxAnalysis and LogicalAnalysis.
    - **Dependency Inversion Principle (DIP)**: High-level modules should not depend on low-level modules; both should depend on abstractions.
        - Example: The BackendClient depends on an abstract AnalysisService interface, not a specific implementation.
2. **Unit Testing:**
    - Use **JUnit** for Java (IDE plugin) and **Pytest** for Python (backend) to write unit tests.
    - Aim for **80%+ test coverage** to ensure robustness.
    - Example: Test cases for SyntaxAnalyzer to verify correct detection of missing semicolons.
3. **Continuous Integration and Deployment (CI/CD):**
    - Use **GitHub Actions** or **Jenkins** for CI/CD pipelines.
    - Automate the following processes:

- **Code Linting**: Ensure code follows style guidelines (e.g., PEP 8 for Python, Google Java Style Guide).
- **Unit Testing**: Run tests automatically on every commit.
- **Integration Testing**: Verify interactions between subsystems.
- **Deployment**: Automatically deploy updates to staging or production environments.

4. **Code Reviews:**
   - Conduct **peer code reviews** to ensure adherence to standards and identify potential issues.
   - Use **pull requests** with mandatory reviews before merging into the main branch.

1.3.3. Summary of Standards

By adhering to these engineering standards, SyntaxSavior will achieve:

- **Clear and comprehensive documentation** (IEEE 1016-2009, UML 2.5).
- **High-quality, maintainable code** (SOLID principles, unit testing).
- **Efficient development workflows** (CI/CD pipelines, code reviews).

These standards ensure that the system is **robust**, **scalable**, and **easy to maintain**, meeting both educational and technical goals.

1.4. Definitions, acronyms, and abbreviations

1.4.1. Core Definitions

1. **SyntaxSavior**:
   - The educational assistance system designed to help students in introductory programming courses by providing real-time, context-aware feedback on their code.
2. **IDE Plugin**:
   - A software extension integrated into a development environment (e.g., Eclipse, VS Code) that provides real-time code analysis and feedback.
3. **Backend Server**:
   - The central processing unit of SyntaxSavior, responsible for analyzing code, generating feedback, and managing interactions with the AI model and database.

4. **AI Model**:
   - A machine learning model fine-tuned to provide educational feedback based on student code and course materials.
5. **Vector Database**:
   - A database optimized for storing and retrieving high-dimensional vector embeddings, used for similarity-based searches (e.g., finding relevant course materials).
6. **Surface-Level Feedback**:
   - Immediate, automated feedback on basic errors (e.g., syntax errors, missing brackets) provided by the IDE plugin.
7. **In-Depth Analysis**:
   - Detailed feedback on logical and runtime errors, generated by the backend server using the AI model.
8. **Lab Task**:
   - A specific programming assignment given to students during laboratory sessions, aligned with the course curriculum.

1.4.2. Acronyms and Abbreviations

1. **API**: Application Programming Interface
   - A set of protocols and tools for building software applications, used for communication between the IDE plugin and backend server.
2. **REST**: Representational State Transfer
   - A software architectural style used for designing networked applications, chosen for its simplicity and scalability.
3. **UML**: Unified Modeling Language
   - A standardized modeling language used to visualize the design of a system (e.g., class diagrams, sequence diagrams).
4. **SOLID**:
   - A set of five design principles for writing maintainable and scalable code:
     - **S**: Single Responsibility Principle
     - **O**: Open/Closed Principle
     - **L**: Liskov Substitution Principle
     - **I**: Interface Segregation Principle
     - **D**: Dependency Inversion Principle

5. **CI/CD**: Continuous Integration and Continuous Deployment
    - A set of practices and tools for automating the integration, testing, and deployment of code changes.
6. **JUnit**:
    - A unit testing framework for Java, used to test the IDE plugin.
7. **LLM**: Large Language Model
    - A machine learning model trained on large datasets to understand and generate human-like text, used for providing educational feedback.
8. **AST**: Abstract Syntax Tree
    - A tree representation of the structure of source code, used for analyzing and transforming code.
9. **FAQ**: Frequently Asked Questions
    - A collection of common questions and answers, stored in the vector database for quick retrieval.

1.4.3. Assumptions

1. **Development Environment**:
    - The IDE plugin will initially target a single environment, IDE or text editor, but the design is flexible enough to support other IDEs or text editors like **VS Code** or **IntelliJ IDEA** or **Eclipse** later down the line.
2. **AI Model Integration**:
    - The AI model will be fine-tuned from an existing pre-trained model (e.g., GPT, Codex) rather than trained from scratch.
3. **Vector Database**:
    - The vector database will use **Milvus** or **ChromaDB** for efficient similarity searches.
4. **Error Classification**:
    - Errors will be classified into three categories: **syntax**, **runtime**, and **logical**.

2. Packages

2.1. IDE Plugin Package

Responsibilities:

- Provides real-time code monitoring and surface-level error detection.
- Acts as the primary interface between the student and the SyntaxSavior system.

Key Components:

1. **CodeMonitor**: Observes code changes in the IDE and triggers analysis.
2. **SyntaxAnalyzer**: Detects surface-level errors (e.g., syntax errors, missing brackets).
3. **FeedbackRenderer**: Displays feedback and highlights errors in the IDE.
4. **BackendClient**: Communicates with the backend server for in-depth analysis.

Dependencies:

- **VS Code Extension API**: For integrating the plugin into Visual Studio Code.
- **REST Client Library**: For sending code snippets to the backend.

Interactions:

- Communicates with the **Backend Processing Package** to send code for in-depth analysis.
- Receives feedback from the backend and displays it to the student.

Considerations:

- The plugin will initially target **VS Code** due to its popularity and extensibility.
- Support for other IDEs (e.g., Eclipse, IntelliJ) can be added in future iterations.

2.2. Backend Processing Package

Responsibilities:

- Analyzes student code for logical and runtime errors.
- Generates contextual feedback using the AI model.
- Manages interactions with the vector database.

Key Components:

1. **RequestHandler**: Validates and routes incoming requests.
2. **CodeAnalysisEngine**: Performs code analysis using AST parsing and rule-based checks.
3. **LanguageModelInterface**: Connects to the AI model for generating feedback.
4. **VectorDBManager**: Retrieves relevant course materials from the vector database.

Dependencies:

- **Spring Framework**: For building the backend server (preferred due to its robustness and Java compatibility).
- **Alternative Options**:
  - **Flask (Python)**: Lightweight and easy to use, but less suitable for Java-based projects.
  - **FastAPI (Python)**: High performance, but requires additional effort to integrate with Java components.

Interactions:

- Receives code snippets from the **IDE Plugin Package**.
- Sends feedback and explanations back to the plugin.
- Queries the **Vector Database Package** for relevant course materials.

Considerations:

- Horizontal scaling (e.g., Kubernetes) was considered but deemed unnecessary due to the project's small scale.

2.3. AI Model Integration Package

Responsibilities:

- Provides educational feedback based on student code and course materials.
- Fine-tunes a pre-trained model for Java-specific tasks.

Key Components:

1. **ModelTrainer**: Fine-tunes the pre-trained model on Java programming tasks.
2. **InferenceService**: Generates hints and explanations using the fine-tuned model.

Dependencies:

- **Deepseek Model**: Likely to be used due to its performance and compatibility with educational tasks.
- **Alternative Options**:
  - **GPT**: Widely available but may require extensive fine-tuning.
  - **Codex**: Specialized for code but less flexible for educational feedback.

Interactions:

- Receives code and task details from the **Backend Processing Package**.
- Sends feedback and explanations back to the backend.

Considerations:

- The model will focus on **Java** initially, with potential support for other languages in the future.

2.4. Vector Database Package

Responsibilities:

- Stores and retrieves course materials, FAQs, and code analysis results.
- Enables similarity-based searches for contextual feedback.

Key Components:

1. **VectorDBManager**: Handles CRUD operations and similarity searches.
2. **DataIngestor**: Converts course materials into vector embeddings.

Dependencies:

- **Milvus** or **ChromaDB**: Likely to be used for efficient vector storage and retrieval.

Interactions:

- Receives queries from the **Backend Processing Package**.
- Returns relevant course materials and examples.

Considerations:

- Real-time updates to the database (e.g., adding new course materials) are not required initially.

3. Class Interfaces

This section will explore the system as presented in the prototype UMLs presented below.

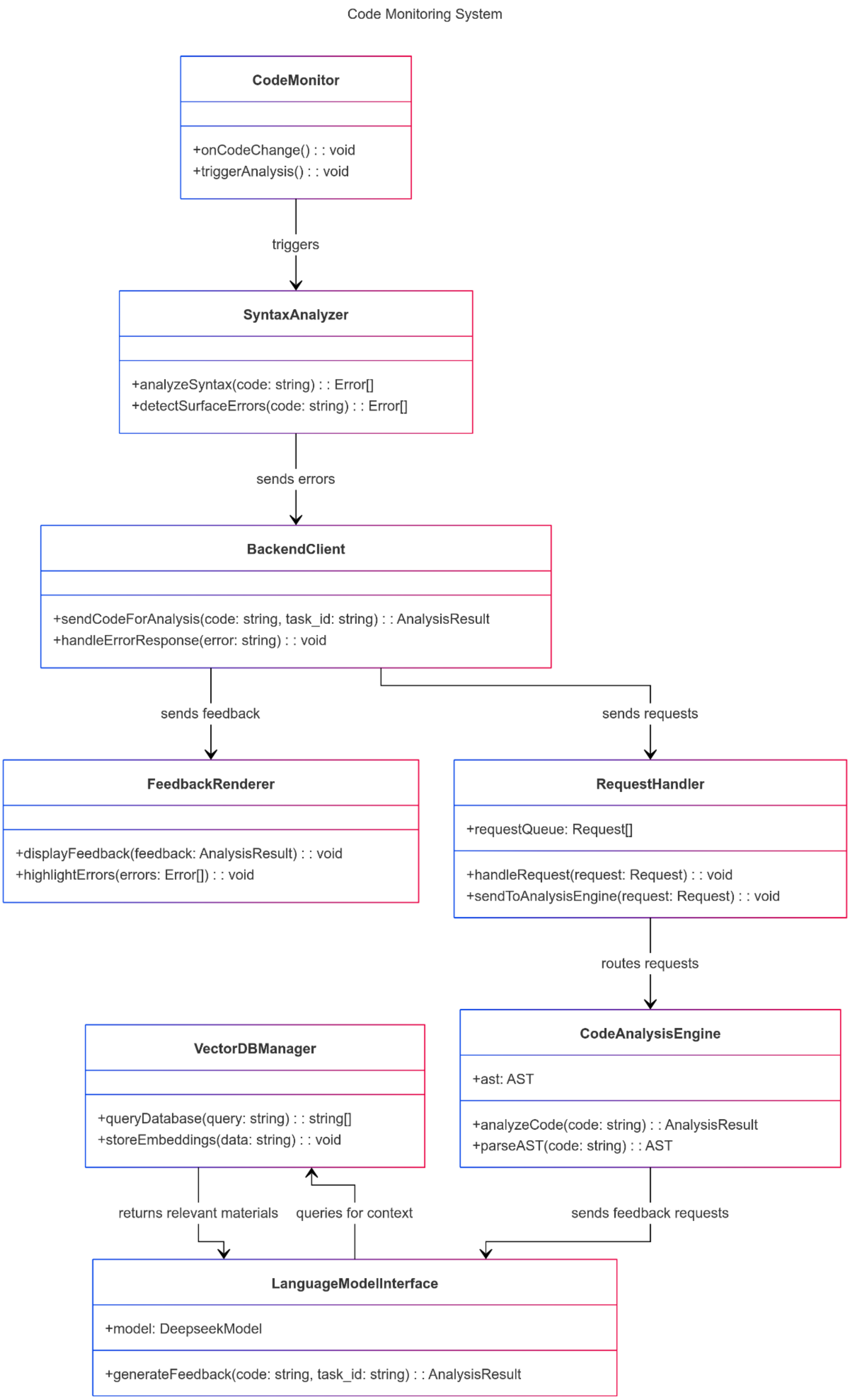

Figure 6 SyntaxSavior Prototype UML

3.1. IDE Plugin Components

3.1.1. CodeMonitor

Responsibilities:

- Observes code changes in the IDE and triggers analysis.

Attributes:

- code: string: The current code snippet.
- lastChangeTime: datetime: Timestamp of the last code change.

Methods:

- onCodeChange(): void: Listens for code changes and triggers analysis.
- triggerAnalysis(): void: Sends the code to the SyntaxAnalyzer for surface-level analysis.

Relationships:

- Uses the **Observer pattern** to monitor code changes.
- Calls SyntaxAnalyzer.analyzeSyntax() for surface-level analysis.

3.1.2. SyntaxAnalyzer

Responsibilities:

- Detects surface-level errors (e.g., syntax errors, missing brackets).

Attributes:

- errorList: Error[]: List of detected errors.

Methods:

- analyzeSyntax(code: string): Error[]: Analyzes the code for surface-level errors.
- detectSurfaceErrors(code: string): Error[]: Detects specific errors (e.g., missing semicolons).

Relationships:

- Called by CodeMonitor for surface-level analysis.
- Sends errors to FeedbackRenderer for display.

3.1.3. FeedbackRenderer

Responsibilities:

- Displays feedback and highlights errors in the IDE.

Attributes:

- feedback: AnalysisResult: The feedback to display.

Methods:

- displayFeedback(feedback: AnalysisResult): void: Displays feedback in the IDE.
- highlightErrors(errors: Error[]): void: Highlights errors in the code editor.

Relationships:

- Receives feedback from SyntaxAnalyzer and BackendClient.

3.2. Backend Components

3.2.1. RequestHandler

Responsibilities:

- Validates and routes incoming requests.

Attributes:

- requestQueue: Request[]: Queue of incoming requests.

Methods:

- handleRequest(request: Request): void: Validates and routes the request.
- sendToAnalysisEngine(request: Request): void: Sends the request to the CodeAnalysisEngine.

Relationships:

- Uses the **Strategy pattern** to handle different types of requests.

3.2.2. CodeAnalysisEngine

Responsibilities:

- Analyzes code for logical and runtime errors.

Attributes:

- ast: AST: Abstract Syntax Tree representation of the code.

Methods:

- analyzeCode(code: string): AnalysisResult: Analyzes the code and generates feedback.
- parseAST(code: string): AST: Parses the code into an AST.

Relationships:

- Called by RequestHandler for code analysis.
- Sends feedback to LanguageModelInterface for contextual explanations.

3.2.3. LanguageModelInterface

Responsibilities:

- Connects to the AI model for generating feedback.

Attributes:

- model: DeepseekModel: The fine-tuned AI model.

Methods:

- generateFeedback(code: string, task_id: string): AnalysisResult: Generates feedback using the AI model.

Relationships:

- Called by CodeAnalysisEngine for feedback generation.

3.3. Error Handling Scenarios

3.3.1. Error Class Hierarchy

Responsibilities:

- Represents different types of errors (syntax, runtime, logical).

Attributes:

- type: string: The type of error (e.g., "syntax", "runtime").
- message: string: A user-friendly error message.
- line: int: The line number where the error occurred.

Methods:

- getHint(): string: Returns a hint for resolving the error.

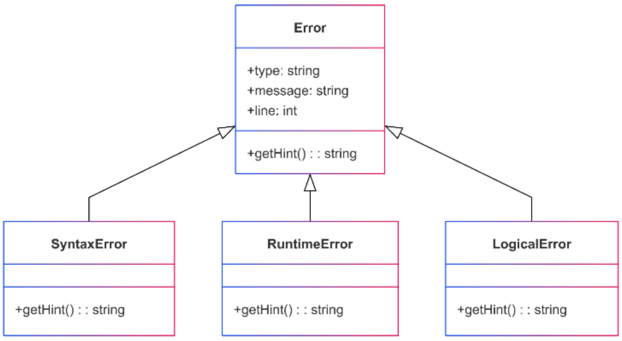

Figure 7 Error Classes Hierarchy UML

3.3.2. Sequence Diagram for Error Handling

Scenario of Sequence: A student submits code with a syntax error, and the system provides feedback.

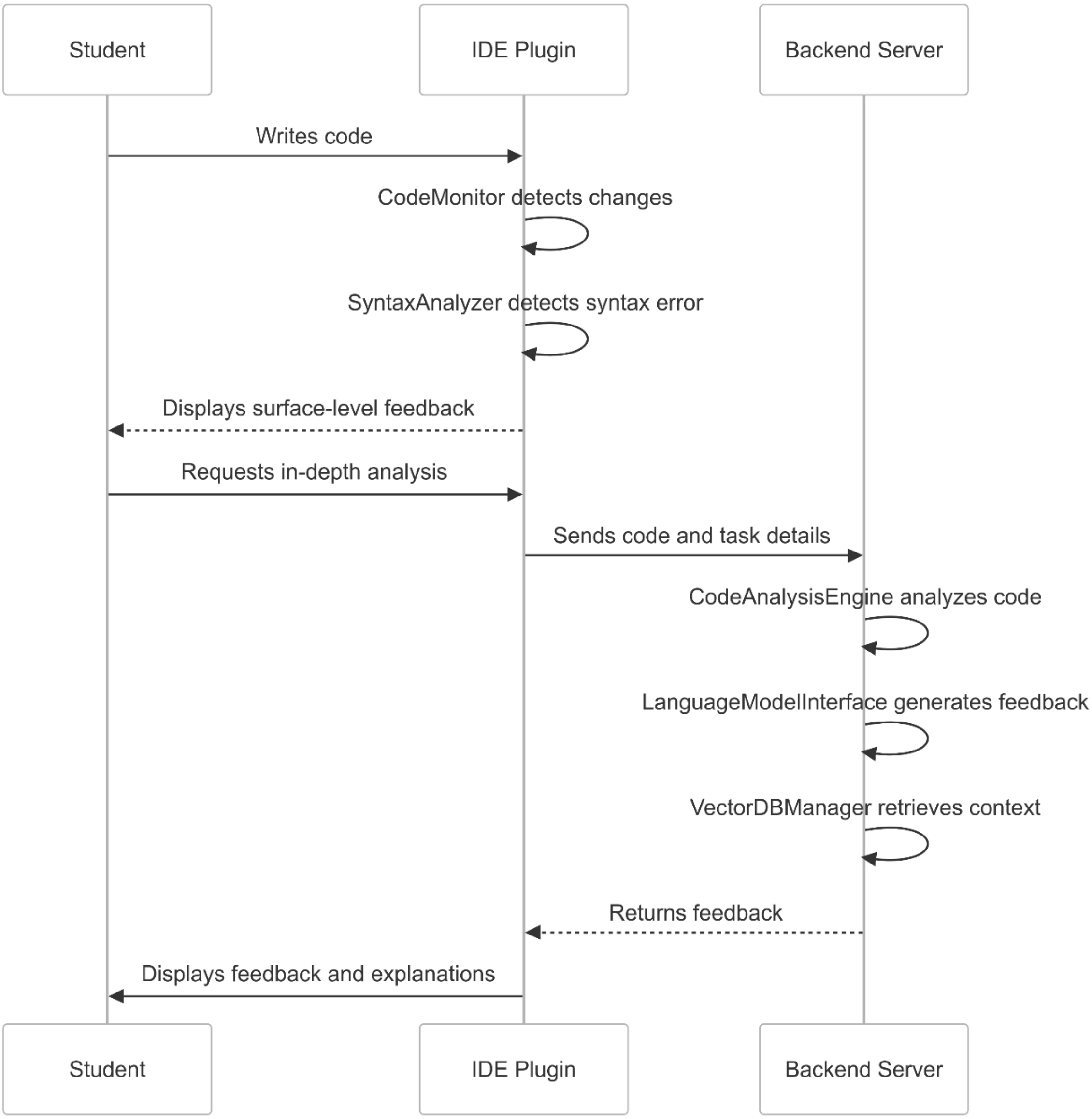

Figure 8 Sequence Diagram for Error Handling

4. Glossary

- **Abstract Syntax Tree (AST)**: A tree representation of the structure of source code, used for analyzing and transforming code.
- **Access Control**: Processes that manage who can access specific data and system resources.

- **AI Model**: A machine learning model integrated into the backend to analyze student code and provide educational feedback.
- **API (Application Programming Interface)**: A set of protocols and tools for building software applications, used for communication between the IDE plugin and backend server.
- **Backend**: The central server that processes user requests, handles code analysis, and manages data storage.
- **Bidirectional Data Communication**: The two-way flow of data between frontend and backend systems.
- **ChromaDB**: An open-source vector database for managing embeddings.
- **CI/CD (Continuous Integration and Continuous Deployment)**: A set of practices and tools for automating the integration, testing, and deployment of code changes.
- **Code Analysis Agent**: An AI-powered agent that analyzes user-submitted code and sends findings to the server for processing.
- **Code Analysis Engine**: A backend component that analyzes code for logical, structural, and syntactical issues.
- **CodeMonitor**: A frontend component that observes code changes in the IDE and triggers analysis.
- **Curriculum Database**: A repository of course materials, syllabi, and explanations of programming concepts.
- **Data Display Manager**: A frontend component responsible for visualizing analysis results and feedback.
- **Data Ingestion**: The process of converting data into embeddings for storage in the vector database.
- **Deepseek Model**: A pre-trained AI model fine-tuned for generating educational feedback.
- **Embedding**: A high-dimensional vector representation of data (e.g., text, code) used for similarity-based searching.
- **Error Handling**: Processes that ensure issues are identified and resolved to minimize system disruption.
- **Event Dispatcher**: Coordinates user interactions and plugin-specific events.
- **FeedbackRenderer**: A frontend component that displays feedback and highlights errors in the IDE.

- **Frontend Interface**: The user-facing part of the system that displays code analysis results, course materials, and facilitates communication between the plugin and backend.
- **IDE Plugin**: A software extension integrated into a development environment (e.g., VS Code) to provide real-time syntax checking, code analysis, and feedback.
- **IEEE 1016-2009**: Standard for System Design Documentation.
- **Informational Panel Plugin**: A non-communicative chatbot providing quick access to frequently sought information.
- **JUnit**: A unit testing framework for Java, used to test the IDE plugin.
- **Language Model Interface**: A backend component that connects to the AI model for generating feedback.
- **Learning Management System (LMS)**: A platform for hosting course content, assignments, and managing student-instructor communication.
- **LogicalError**: A type of error representing mistakes in the logic of the code.
- **Milvus**: A vector database used for efficient similarity searches.
- **Plagiarism Detection**: Mechanism for identifying code similarity to detect potential academic dishonesty.
- **Pytest**: A testing framework for Python, used to test the backend server.
- **RequestHandler**: A backend component that validates and routes incoming requests.
- **REST (Representational State Transfer)**: A software architectural style used for designing networked applications.
- **Role-Based Access Control (RBAC)**: A method to restrict access based on user roles within the system.
- **RuntimeError**: A type of error that occurs during the execution of the code.
- **Safe-Exam-Browser (SEB)**: A secure browser for online assessments.
- **Security Services**: Measures like encryption and multi-factor authentication to protect system data.
- **Server Overload Management**: Strategies like load balancing and scaling to handle high traffic.
- **Session Manager**: Tracks and manages user sessions across the platform.
- **Similarity Search**: A query process for finding semantically similar data in the vector database.
- **SOLID Principles**: A set of five design principles for writing maintainable and scalable code.

- **State Manager**: Maintains the current state of the frontend application.
- **SyntaxAnalyzer**: A frontend component that detects surface-level errors in the code.
- **SyntaxError**: A type of error representing mistakes in the syntax of the code.
- **SyntaxSavior**: The system designed to assist students in learning programming through real-time feedback, code analysis, and course resources.
- **Topic Retrieval Agent**: An AI agent that retrieves relevant course topics based on code analysis findings.
- **UML (Unified Modeling Language)**: A standardized modeling language used to visualize the design of a system.
- **User Authentication**: The process of verifying user identity before granting access.
- **Vector Database**: A database storing data in vector form, enabling fast similarity searches based on embeddings.
- **VectorDBManager**: A backend component that handles CRUD operations and similarity searches in the vector database.
- **Virtual Programming Lab (VPL)**: An automated grading system integrated with the LMS for code submission evaluation.
- **VS Code Extension API**: The API used to develop the IDE plugin for Visual Studio Code.
- **Workflow**: The sequence of processes for data ingestion, analysis, and result delivery.

5. References

1. **CMPE 113: Introduction to Programming Syllabus**
   - TED University, 2024.
   - Provides foundational information on course structure, objectives, and expectations.
2. **IEEE 1016-2021**
   - Standard for System Design Documentation.
   - Website: https://standards.ieee.org/ieee/1016/4502/
3. **UML 2.5**
   - Unified Modeling Language (UML) specification for creating diagrams.
   - Documentation: https://www.omg.org/spec/UML
4. **SOLID Principles**
   - A set of design principles for writing maintainable and scalable code.
   - Reference: https://en.wikipedia.org/wiki/SOLID
5. **RESTful API Design**

    - Guidelines for designing RESTful APIs.
    - Reference: https://restfulapi.net
6. **VS Code Extension API**
    - The API used to develop the IDE plugin for Visual Studio Code.
    - Documentation: https://code.visualstudio.com/api
7. **Artificial Intelligence in Education**
    - Göçen, A., & Aydemir, F. (2020). Artificial intelligence in education and schools. *Research on Education and Media*, 12(1), 13–21.
    - DOI: https://doi.org/10.2478/rem-2020-0003
8. **Ethical Challenges in AI Education**
    - Akgün, S., & Greenhow, C. (2021). Artificial intelligence in education: Addressing ethical challenges in K-12 settings. *AI and Ethics*, 2(3), 431–440.
    - DOI: https://doi.org/10.1007/s43681-021-00096-7

Additional Resources and Tools Used

1. **Mermaid Live Editor**
    - Used to create and render UML diagrams and flowcharts.
    - Website: www.mermaidchart.com