# **System Documentation**

**Name:** Strimbeanu Mihai Alexandru

**Class:** CEN4.S2A

## **1. Introduction**

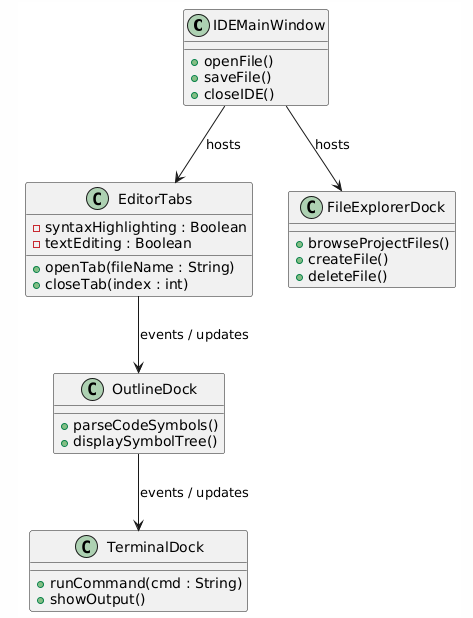
This document provides a comprehensive overview of the custom IDE-like application developed by *Strimbeanu Mihai Alexandru (CEN4.S2A)*.

The system offers:

* A **PySide6-based** GUI layout
* A **File Explorer** panel supporting basic file operations (create, rename, delete)
* A **Tabbed Code Editor** with syntax highlighting for Python and C
* An **Outline** panel that displays code symbols (functions, classes, etc.)
* An **Integrated Terminal** to run commands and view output
* A **Menu Bar** with essential file, edit, view, and run features

This documentation details the application’s architecture, features, and implementation details.

## **2. High-Level Architecture**



### **Major Components**

1. **Main Window (IDEMainWindow)**
2. The top-level controller managing layout, docks, and the menu bar.

Hosts references to the editor tabs, file explorer, outline, and terminal docks.

1. **File Explorer (FileExplorerDock)**
   * Dock on the left, allowing directory navigation and file actions (open, rename, etc.).
   * Sends open-file events to the editor.
2. **Tabbed Code Editor (EditorTabs)**
   * Manages multiple open files in tabs.
   * Invokes Python/C syntax highlighting based on file extension.
   * Allows saving, closing, and pinning tabs.
3. **Outline Panel (OutlineDock)**
   * Displays hierarchical symbols (classes, functions) extracted from the active file.
   * Clicking a symbol navigates the editor to its line.
4. **Terminal (TerminalDock)**
   * Dock at the bottom with a text output view and command input line.
   * Runs commands (e.g., Python scripts, GCC builds) via QProcess.
   * Outputs merged stdout/stderr.
5. **Syntax Highlighting**
   * **Python**: tokenize-based approach with optional second pass for advanced features.
   * **C**: Regex-based highlighting for keywords, function calls, comments, numbers, etc.

## **3. Key Features**

1. **File Explorer**
   * Navigate directories
   * Right-click context menu to create, rename, or delete files/folders
   * Double-click to open files in the editor
2. **Tabbed Editor**
   * Multiple files open at once
   * Syntax highlighting (Python .py and C .c)
   * Basic editing commands (cut, copy, paste, undo, redo)
   * Pin tabs to prevent accidental closure
3. **Outline**
   * Parses Python or C files to find top-level symbols (classes, functions, preprocessor directives, etc.)
   * Refreshes on tab switch or file save
   * Clicking a symbol jumps the editor’s cursor to that line
4. **Terminal**
   * Embedded command line using QProcess
   * Allows running arbitrary shell commands (like python, gcc, git)
   * Displays merged stdout and stderr output
   * “Refresh Terminal” feature clears and re-initializes the terminal
5. **Menu Bar**
   * **File**: Open Folder, Save, Exit
   * **Edit**: Cut, Copy, Paste, Undo, Redo
   * **Run**: Execute current Python or C file
   * **View**: Toggle visibility of File Explorer, Outline, Terminal
   * **Settings**: Placeholder (e.g., for preferences)

## **4. System Flows (Use Cases)**

1. **Open a Folder**
   * User selects a directory from “File > Open Folder.”
   * FileExplorerDock updates its root path to the chosen directory.
   * The directory tree is displayed in the explorer.
2. **Open a File**
   * User double-clicks a file in the file explorer.
   * EditorTabs.open\_file() loads the file, applies syntax highlighting based on the extension.
3. **Edit and Save Files**
   * User edits in the QPlainTextEdit.
   * Pressing Ctrl+S (or “File > Save”) writes changes to disk.
   * Outline can be refreshed if the file structure changed.
4. **Outline Navigation**
   * When the active tab changes, OutlineDock.refresh\_outline() re-parses the file.
   * Clicking on a symbol in the outline sets the editor’s cursor to that line.
5. **Run Code**
   * If active file is a .py file, the IDE runs python file.py in the terminal.
   * If active file is a .c file, it runs gcc file.c -o file.exe && .\file.exe (Windows) or similar.
   * The terminal displays command output interactively.
6. **Terminal Commands**
   * User can manually enter commands (e.g., git status, dir, etc.) and see outputs.

## **5. Implementation Details**

### **5.1 Technologies**

* **Python 3**
* **PySide6** (Qt for Python)
* **Syntax Highlighters**:
  + Python: Built-in tokenize
  + C: Regex-based with naive multiline comment handling
* **File Explorer**: QFileSystemModel or custom logic
* **Terminal**: QProcess capturing merged stdout/stderr

### **5.2 Project Structure**

IDE\_Project/

main.py # Entry point

mainwindow.py # IDEMainWindow: sets up docks, menu, layout

editor.py # EditorTabs + syntax highlighter classes

fileexplorer.py # FileExplorerDock

outline.py # OutlineDock: parses Python/C for symbols

terminal.py # TerminalDock: handles QProcess commands

### **5.3 Editor Behavior**

* Every open file is a QPlainTextEdit in EditorTabs.
* \_apply\_highlighting(editor, file\_path) picks Python or C highlighter.
* Tab key can be set to insert spaces or keep \t; setTabStopDistance() sets visual width.

### **5.4 Outline Logic**

* **Python**: Uses ast or tokenize to find def and class plus line numbers.
* **C**: Uses custom regex or partial parser for function definitions, macros, preprocessor lines.
* Clicking an item moves the editor’s cursor using QTextCursor to that line.

### **5.5 Terminal Logic**

* A QLineEdit for command input, QPlainTextEdit for output.
* On Enter or programmatic execute\_command(cmd), the system starts a QProcess.
* Output is appended to the text area, and a new prompt appears upon completion.

## **6. Deployment and Setup**

1. **Install Python 3.9+**
2. **Install PySide6 and denepdencies**
3. **Run**

## **7. Known Limitations**

* **Python** highlighting is limited by tokenize; advanced function/argument coloring requires an extra pass or a more robust parser.
* **C** highlighting uses naive regex; multiline macros, advanced pointer syntax, or complex definitions may break.
* **No real-time Outline**: Outline updates primarily on tab switch or manual refresh.
* **No debugging**: Currently no integrated stepping or breakpoints.

## **8. Future Enhancements**

1. **More Advanced Parsing**
   * Use a *concrete syntax tree* library (e.g., LibCST) for Python, or more advanced solutions for C.
2. **Autocomplete / IntelliSense**
   * Integrate with a Language Server Protocol for code completion and error checking.
3. **Debugging Interface**
   * Add breakpoints, stepping, variable watch windows.
4. **Real-Time Outline**
   * Continuously re-parse as the user types, possibly with a slight delay to avoid performance issues.
5. **Version Control**
   * Integrate Git commands in the IDE with a dedicated panel or tab.

## **9. Conclusion**

This IDE project demonstrates a functional yet extensible code editor with:

* **Multi-file editing**
* **File system navigation**
* **Symbol-based outline**
* **Terminal command execution**

While not as feature-complete as large commercial IDEs, it provides a solid foundation for advanced extensions in parsing, debugging, and project management.