GitHub Repository Refactoring Analyzer - Complete Code Analysis

🏗️ Architecture Overview

This is a **Streamlit-based web application** that analyses GitHub repositories and provides AI-powered refactoring suggestions. The system consists of 4 main Python files that work together:

app.py (Main UI)
  ↓ calls
github_refactor_analyzer.py (GitHub API Handler)
  ↓ passes data to
gemini_refactor_engine.py (AI Analysis Engine)
  ↓ results processed by
refactor_report_generator.py (Report Generator)

📄 File-by-File Analysis

1. app.py - Main Application & User Interface

Purpose: Streamlit web interface that orchestrates the entire refactoring analysis process

Key Functions:

main () - The core application function

- Sets up the Streamlit page configuration
- Creates the sidebar with API key inputs and analysis options
- Handles the main UI layout with repository input
- Orchestrates the entire analysis workflow
- Manages progress tracking and error handling

UI Configuration Functions:

- **Sidebar Setup**:
  - API key inputs (Gemini API, GitHub token)
  - Analysis focus checkboxes (performance, maintainability, design patterns, etc.)
  - Analysis scope settings (max files, include tests, focus on large files)

Analysis Workflow (triggered by "Analyze Repository" button):

1. **Input Validation**: Checks GitHub URL and API keys
2. **Client Initialization**: Sets up GitHub and Gemini clients
3. **Repository Validation**: Verifies the repository exists
4. **File Fetching**: Downloads repository structure and files
5. **AI Analysis**: Sends data to Gemini for refactoring suggestions
6. **Results Storage**: Saves results in Streamlit session state

Display Functions - These render the analysis results in tabs:

display_performance_suggestions()

- Shows performance optimization recommendations
- Displays before/after code examples
- Includes performance scores and metrics

display_maintainability_suggestions()

- Shows code maintainability improvements
- Displays maintainability metrics (complexity, duplicates)
- Shows current vs improved code approaches

display_design_pattern_suggestions()

- Shows design pattern recommendations
- Lists existing patterns found
- Suggests better architectural patterns

display_code_quality_suggestions()

- Shows code quality enhancements
- Identifies code smells and style issues
- Shows problematic vs improved code

display_security_suggestions()

- Shows security vulnerability recommendations
- Displays risk levels and vulnerability types
- Shows vulnerable vs secure code implementations

display_modularity_suggestions()

- Shows modularity improvements
- Displays cohesion and coupling metrics
- Suggests better code organization

display_summary_tab()

- Provides overall analysis summary
- Shows suggestion counts by category
- Creates priority recommendations

display_export_tab()

- Handles report generation in multiple formats
- Provides download buttons for PDF, Excel, Markdown
- Shows JSON export option

---

2. github_refactor_analyzer.py - GitHub Repository Handler

Purpose: Handles all GitHub API interactions and repository analysis

Global Variables:

- github_session: Requests session for GitHub API calls
- github_headers: Authentication headers for API requests
- REFACTOR_EXTENSIONS: Supported file types for analysis
- PRIORITY_REFACTOR_FILES: High-priority files (package.json, requirements.txt, etc.)
- SKIP_REFACTOR_PATTERNS: Files/folders to ignore
- TEST_PATTERNS: Test file identification patterns

Key Functions:

initialize_clients(gemini_api_key, github_token)

- Sets up the GitHub API client with authentication
- Configures request headers with optional GitHub token
- Prepares the session for API calls

validate_repository(github_url)

- Extracts owner/repo name from GitHub URL using regex
- Makes GitHub API call to verify repository exists
- Returns repository metadata (name, description, language, stars, etc.)
- Returns None if repository is invalid or inaccessible

should_skip_file_for_refactoring(path, include_tests)

- Determines if a file should be excluded from analysis
- Checks against skip patterns (node_modules, .git, dist, etc.)
- Conditionally excludes test files based on user preference
- Filters out binary and media files

get_file_complexity_score(content, file_type)

- Calculates a complexity score (0-100) for code files
- Base score from file length (lines / 10, max 50 points)
- Adds points for complexity indicators (functions, classes, control structures)
- Uses language-specific patterns for accurate scoring
- Different patterns for JS/TS (function, class, if, for, while) vs Python (def, class, if, for, while) etc.

fetch_repository_contents_recursive(owner, repo_name, path)

- Recursively fetches all files from a GitHub repository
- Uses GitHub Contents API to traverse directory structure
- Skips directories that match skip patterns
- Returns list of all file metadata

fetch_repository_for_refactoring(github_url, options)

- **Main orchestration function** for repository analysis
- Fetches all repository files recursively
- Filters files based on:
  - File extensions (code files only)
  - Priority files (config files like package.json)
  - User options (include tests, config files)
- **File Prioritization Logic**:
  - Priority files first (package.json, requirements.txt)
  - Then by size (large files first if focus_large_files=True)
- **Downloads file contents** for selected files
- **Calculates complexity scores** for each file
- **Limits analysis** to max_files parameter (default 30, increased to 60 for analysis)
- Returns structured data with:
  - analyzed_files: File contents and metadata
  - statistics: Analysis statistics
  - files: File list with metadata

get_language_from_extension(ext)

- Maps file extensions to programming language names
- Used for syntax highlighting and language-specific analysis

extract_functions_and_classes(content, file_type)

- Parses code to extract function and class definitions
- Uses regex patterns specific to each programming language
- Returns list of code entities with line numbers and types

---

3. gemini_refactor_engine.py - AI Analysis Engine

Purpose: Handles all AI-powered analysis using Google's Gemini 2.5 Pro model

Global Variables:

- gemini_client: Global Gemini API client instance

Key Functions:

initialize_gemini(api_key)

- Initializes the global Gemini client with API key
- Sets up connection to Google's Gemini 2.5 Pro model

build_refactoring_prompt(repo_structure, analysis_options, repo_info)

- **Most complex function** - builds comprehensive AI prompt
- **Repository Overview Section**:
  - Repository metadata (name, language, description)
  - Codebase statistics (file counts, languages, complexity)
- **File Contents Section**:
  - Sorts files by priority and complexity
  - Includes top 50 files for analysis
  - Truncates very long files (>20KB) to stay within token limits
  - Formats each file with metadata and syntax highlighting
- **Analysis Focus Areas**:
  - Dynamically includes only selected analysis types
  - Each focus area has specific instructions (performance, maintainability, etc.)
- **JSON Structure Definition**:
  - Defines expected output format for each analysis type
  - Detailed schema for performance, maintainability, design patterns, etc.
  - Ensures structured, parseable responses from AI

generate_refactor_suggestions(repo_structure, analysis_options, repo_info)

- **Main AI interaction function**
- Calls build_refactoring_prompt() to create the prompt
- Makes API call to Gemini 2.5 Pro with:
  - Model: "gemini-2.5-pro"
  - Response format: JSON
  - Temperature: 0.3 (focused, less random)
  - Max tokens: 100,000 (for comprehensive analysis)
- **Error Handling**:
  - Catches JSON parsing errors
  - Returns fallback structure with error details

    - Logs errors for debugging
- Returns parsed JSON with refactoring suggestions

---

4. refactor_report_generator.py - Report Generation System

Purpose: Generates comprehensive reports in multiple formats (Markdown, PDF, Excel)

Key Functions:

generate_markdown_report(suggestions, repo_info, analysis_options)

- **Main report generation function**
- Creates comprehensive markdown report with:
    - Executive summary with repository info
    - Dynamic table of contents based on selected analyses
    - Detailed sections for each analysis type
    - Summary with implementation recommendations
- Uses helper functions for each section type
- Returns complete markdown string

Section Generator Functions:

generate_performance_section(performance_data)

- Formats performance suggestions into markdown
- Shows performance scores and metrics
- Includes before/after code examples with syntax highlighting
- Displays priority levels and impact descriptions

generate_maintainability_section(maintainability_data)

- Formats maintainability suggestions
- Shows maintainability metrics (complexity, duplication)
- Displays current vs improved approaches
- Lists benefits of each improvement

generate_design_patterns_section(design_patterns_data)

- Shows existing patterns found in codebase
- Suggests better architectural patterns
- Includes implementation examples
- Displays complexity levels for each suggestion

generate_code_quality_section(code_quality_data)

- Shows code quality improvements
- Displays quality scores and issue counts
- Shows problematic vs improved code
- Explains why each improvement matters

generate_security_section(security_data)

- Shows security vulnerability recommendations
- Displays risk levels and vulnerability types
- Shows vulnerable vs secure implementations
- Provides step-by-step mitigation instructions

generate_modularity_section(modularity_data)

- Shows modularity improvements
- Displays cohesion/coupling metrics
- Suggests better code organization
- Shows refactoring examples

generate_summary_section(suggestions, analysis_options)

- Creates overall summary with statistics
- Counts suggestions by priority level
- Provides recommended implementation order
- Suggests next steps for improvement

Export Functions:

generate_pdf_report(suggestions, repo_info, analysis_options)

- Converts markdown report to PDF using ReportLab
- Handles different heading levels and formatting
- Falls back to text if ReportLab not available
- Returns PDF as BytesIO buffer for download

generate_excel_report(suggestions, repo_info)

- Creates multi-sheet Excel workbook
- Summary sheet with repository info
- Separate sheets for each suggestion category
- Organizes suggestions in tabular format
- Returns Excel file as BytesIO buffer

🔄 Data Flow & Integration

Complete Workflow:

1. **User Input** (app.py):
   - User enters GitHub URL and selects analysis options
   - API keys configured in sidebar
2. **Repository Validation** (github_refactor_analyzer.py):
   - validate_repository() checks if repo exists
   - Returns repository metadata
3. **File Fetching** (github_refactor_analyzer.py):
   - fetch_repository_for_refactoring() downloads repository structure
   - Filters and prioritizes files based on user options
   - Downloads file contents and calculates complexity scores
4. **AI Analysis** (gemini_refactor_engine.py):
   - build_refactoring_prompt() creates comprehensive analysis prompt
   - generate_refactor_suggestions() sends prompt to Gemini 2.5 Pro
   - Returns structured JSON with suggestions
5. **Results Display** (app.py):
   - Results stored in Streamlit session state
   - Multiple display functions render different suggestion types
   - Tabbed interface for easy navigation
6. **Report Generation** (refactor_report_generator.py):

    - User can export results in multiple formats
    - Markdown, PDF, and Excel reports available
    - Each format optimized for different use cases

Analysis Types:

- **Performance**: Slow algorithms, inefficient loops, memory issues
- **Maintainability**: Complex functions, code duplication, readability
- **Design Patterns**: Architectural improvements, SOLID principles
- **Code Quality**: Code smells, error handling, documentation
- **Security**: Vulnerabilities, authentication issues, data exposure
- **Modularity**: Coupling, cohesion, separation of concerns

Smart File Processing:

- **Priority System**: Config files analyzed first
- **Complexity Scoring**: Files ranked by algorithmic complexity
- **Size Limits**: Large files truncated to stay within AI token limits
- **Language Support**: 20+ programming languages supported
- **Selective Analysis**: User controls what types of files to include

Comprehensive Reporting:

- **Interactive Web Interface**: Real-time results with code examples
- **Multiple Export Formats**: Markdown, PDF, Excel
- **Detailed Explanations**: Before/after code with explanations
- **Prioritized Recommendations**: High/medium/low priority classification
- **Implementation Guidance**: Step-by-step improvement suggestions

This system effectively combines GitHub API integration, AI-powered analysis, and comprehensive reporting to provide actionable refactoring recommendations for any GitHub repository.

Analysis Focus

- Performance optimization: Analyse loops, algorithms, and API calls
- Code Maintainability: Suggest improvements for readability and structure
- Design Pattern: Identify better architectural patterns
- Code Quality: Find code smells and best practices
- Security Issue: Identify potential security vulnerabilities
- Modularity Improvement: Suggest better separation of concerns

Problems and solution to improvement

- Add complexity pattern for complexity score
- I set 60 Files for analyse Need to research on it
- Need to do more preprocessing for better analysis
- Prompt tuning
- Need Human Evaluation for performance matrix

Why GitHub Token is Used:

1. **Higher Rate Limits** - Without a token, GitHub API allows only 60 requests per hour. With a token, you get 5,000 requests per hour, which is essential when analysing large repositories with many files.
2. **Access to Private Repositories** - If you want to analyze your private repositories, you need authentication via a personal access token.
3. **Better Reliability** - Authenticated requests are less likely to be rate-limited or blocked, ensuring the analysis completes successfully.
4. **No Cost** - GitHub personal access tokens are completely free to create and use.

Input Tokens (sent to Gemini):

Base Prompt Structure: ~2,000 tokens

- Repository overview, instructions, JSON schema

Per File Analysis: ~5,000-6,000 tokens per file

- File metadata: ~100 tokens
- File content: Up to 20KB = ~5,000 tokens (assuming 4 chars per token)

Total Input for 50 Files:

- Base prompt: 2,000 tokens
- File content: 50 × 5,000 = 250,000 tokens
- **Total Input: ~252,000 tokens**

Output Tokens (Gemini response):

- **Maximum allowed: 100,000 tokens**

Gemini 2.5 Pro Limits:

- **Context window: 1,000,000 tokens** ✅
- **Input + Output combined: ~352,000 tokens** (well within limit!)

What This Means:

You're using about 35% of Gemini's full capacity, which is excellent because:

1. **Safe margin** - No risk of hitting context limits
2. **Room for growth** - Could analyze even more files if needed
3. **Comprehensive output** - 100K output tokens allow for very detailed suggestions
4. **Cost efficient** - Getting maximum value from each API call

Comparison to previous setup:

- **Before**: ~50,000 input + 8,192 output = 58,192 total tokens
- **Now**: ~252,000 input + 100,000 output = 352,000 total tokens
- **Improvement**: 6x more comprehensive analysis!

Readme.md file:

https://github.com/tushararora-dev/Repository_Refactoring_Analyzer/blob/main/readme.md

GitHub URL:

https://github.com/tushararora-dev/Repository_Refactoring_Analyzer