Module Guide for Software Engineering

Team 4, EcoOptimizers

Nivetha Kuruparan
Sevhena Walker
Tanveer Brar
Mya Hussain
Ayushi Amin

March 25, 2025

1 Revision History

Date	Name	Notes
January 17th, 2025	All	Initial draft
March 24th, 2025	Mya Hussain	Added formalization to multiple modules.
March 24th, 2025	Ayushi Amin	Added formalization to multiple modules.
March 25th, 2025	Ayushi Amin	Updated Trace for Functional Requirements.

2 Reference Material

This section records information for easy reference.

2.1 Abbreviations and Acronyms

symbol	description
AC	Anticipated Change
DAG	Directed Acyclic Graph
M	Module
MG	Module Guide
OS	Operating System
R	Requirement
SC	Scientific Computing
SRS	Software Requirements Specification
Software Engineering	Explanation of program name
UC	Unlikely Change
VS	Visual Studio
API	Application Programing Interface
IDE	Integrated Development Environment
AST	Abstract Syntax Tree
CSV	Comma-Separated Values

Contents

List of Tables

List of Figures

3 Introduction

Decomposing a system into modules is a commonly accepted approach to developing software. A module is a work assignment for a programmer or programming team (Parnas et al., 1984). We advocate a decomposition based on the principle of information hiding (Parnas, 1972). This principle supports design for change, because the "secrets" that each module hides represent likely future changes. Design for change is valuable in SC, where modifications are frequent, especially during initial development as the solution space is explored.

Our design follows the rules layed out by Parnas et al. (1984), as follows:

- System details that are likely to change independently should be the secrets of separate modules.
- Each data structure is implemented in only one module.
- Any other program that requires information stored in a module's data structures must obtain it by calling access programs belonging to that module.

After completing the first stage of the design, the Software Requirements Specification (SRS), the Module Guide (MG) is developed (Parnas et al., 1984). The MG specifies the modular structure of the system and is intended to allow both designers and maintainers to easily identify the parts of the software. The potential readers of this document are as follows:

- New project members: This document can be a guide for a new project member to easily understand the overall structure and quickly find the relevant modules they are searching for.
- Maintainers: The hierarchical structure of the module guide improves the maintainers' understanding when they need to make changes to the system. It is important for a maintainer to update the relevant sections of the document after changes have been made.
- Designers: Once the module guide has been written, it can be used to check for consistency, feasibility, and flexibility. Designers can verify the system in various ways, such as consistency among modules, feasibility of the decomposition, and flexibility of the design.

The rest of the document is organized as follows. Section 4 lists the anticipated and unlikely changes of the software requirements. Section 5 summarizes the module decomposition that was constructed according to the likely changes. Section 6 specifies the connections between the software requirements and the modules. Section 7 gives a detailed description of the modules. Section 8 includes two traceability matrices. One checks the completeness of the design against the requirements provided in the SRS. The other shows the relation between anticipated changes and the modules. Section 9 describes the use relation between modules.

4 Anticipated and Unlikely Changes

This section lists possible changes to the system. According to the likeliness of the change, the possible changes are classified into two categories. Anticipated changes are listed in Section 4.1, and unlikely changes are listed in Section 4.2.

4.1 Anticipated Changes

Anticipated changes are the source of the information that is to be hidden inside the modules. Ideally, changing one of the anticipated changes will only require changing the one module that hides the associated decision. The approach adapted here is called design for change.

AC1: The user interface of the plugin. Enhancements may be required to improve usability or accommodate new features. Specific anticipated changes include:

- **Refactoring Suggestions Display**: Updates to how refactoring suggestions are presented, such as side-by-side views of original and refactored code.
- **Theme Support**: Adding compatibility with various VS Code themes, including light and dark modes.
- **Visual Indicators**: Implementing color-coded indicators to highlight the impact of energy savings for each refactoring suggestion.
- **Interactive Elements**: Introducing interactive components like tooltips or progress indicators to guide users during the refactoring process.
- **Customization Options**: Allowing users to configure UI elements, such as adjusting the sensitivity of code smell detection or selecting preferred refactoring styles.

AC2: The VS Code plugin's functionality. Future versions may expand to support more complex refactorings or additional code smells that users can address with minimal setup. Changes may involve adding more customizable user settings.

AC3: The refactorers responsible for detecting and fixing specific code smells. As more code smells are identified and refactoring techniques are developed, new modules may be added or existing ones may evolve. For example:

- **Base Refactorer** : Updates to the base refactorer module to support new refactoring patterns or improved algorithms.
- **Complex List Comprehension** : Adding or modifying the logic for simplifying complex list comprehensions.
- **Long Element Chain** : Refining the logic to handle longer chains of elements and optimize their readability and performance.
- **Long Lambda Function** : Improvements to better handle long lambda functions, making them more efficient and readable.

- **Long Message Chain** : Extending the module's ability to identify and refactor long message chains.
- **Member Ignoring Method** : Enhancements to the module for detecting methods that ignore members, optimizing the code structure.
- **Repeated Calls** : Optimizing detection and handling of repeated function calls to improve performance.
- **String Concatenation in Loop** : Adjusting the refactorer's logic to improve handling of string concatenation within loops.
- **Long Parameter List** : Future extensions to handle complex parameter lists in a more structured manner, perhaps allowing for simplifications.

AC4: The core logic for identifying specific code smells. As the system evolves, new code smells may be added to the system's detection capabilities, necessitating changes to this module.

AC5: The analyzers used to gather metrics and assess code quality.

4.2 Unlikely Changes

The module design should be as general as possible. However, a general system is more complex. Sometimes this complexity is not necessary. Fixing some design decisions at the system architecture stage can simplify the software design. If these decision should later need to be changed, then many parts of the design will potentially need to be modified. Hence, it is not intended that these decisions will be changed.

UC1: Transitioning from VS Code to another IDE. The plugin is tightly integrated with VS Code's API, making such a change complex.

UC2: Fundamental changes to the core logic of code smell detection. The current architecture is designed around widely accepted principles of software quality.

UC3: Changing from a plugin-based architecture to a standalone application. This would require rethinking the entire deployment and user interaction model.

5 Module Hierarchy

This section provides an overview of the module design. Modules are summarized in a hierarchy decomposed by secrets in Table 1. The modules listed below, which are leaves in the hierarchy tree, are the modules that will actually be implemented.

M1: Smell Module

M2: BaseRefactorer Module

M3: MakeStaticRefactorer Module

M4: UseListAccumulationRefactorer Module

M5: UseAGeneratorRefactorer Module

M6: CacheRepeatedCallsRefactorer Module

M7: LongElementChainRefactorer Module

M8: LongParameterListRefactorer Module

M9: LongMessageChainRefactorer Module

M10: LongLambdaFunctionRefactorer Module

M11: Plugin Initiator Module

M12: Backend Communicator Module

M13: Smell Detector Module

M14: File Highlighter Module

M15: Hover Manager Module

M16: Measurements Module

M17: Pylint Analyzer Module

M18: Smell Refactorer Module

M19: Refactor Manager Module

M20: Cache Manager Module

M21: Filter Manager Module

M22: Energy Metrics Module

M23: View Provider Module

M24: Event Manager Module

6 Connection Between Requirements and Design

The design of the system is intended to satisfy the requirements developed in the SRS. In this stage, the system is decomposed into modules. The connection between requirements and modules is listed in Table **??**.

Level 1	Level 2
Hardware-Hiding Module	None
Behaviour-Hiding Module	Smell Module
	BaseRefactorer Module
	MakeStaticRefactorer Module
	UseListAccumulationRefactorer Module
	UseAGeneratorRefactorer Module
	CacheRepeatedCallsRefactorer Module
	LongElementChainRefactorer Module
	LongParameterListRefactorer Module
	LongMessageChainRefactorer Module
	LongLambdaFunctionRefactorer Module
	PluginInitiator Module
	BackendCommunicator Module
	SmellDetector Module
	FileHighlighter Module
	HoverManager Module
	CacheManager Module
	FilterManager Module
Software Decision Module	Measurements Module
	PylintAnalyzer Module
	SmellRefactorer Module
	RefactorManager Module
	EnergyMetrics Module
	ViewProvider Module
	EventManager Module

Table 1: Module Hierarchy

7 Module Decomposition

Modules are decomposed according to the principle of "information hiding" proposed by Parnas et al. (1984). The *Secrets* field in a module decomposition is a brief statement of the design decision hidden by the module. The *Services* field specifies *what* the module will do without documenting *how* to do it. For each module, a suggestion for the implementing software is given under the *Implemented By* title. If the entry is *OS*, this means that the

module is provided by the operating system or by standard programming language libraries. *Software Engineering* means the module will be implemented by the Software Engineering software.

Only the leaf modules in the hierarchy have to be implemented. If a dash (–) is shown, this means that the module is not a leaf and will not have to be implemented.

7.1 Hardware Hiding Modules

This system has no hardware components.

7.2 Behaviour-Hiding Module

7.2.1 Smell Module (M1)

Secrets: Data structure of a code smell.

Services: Provides an interface for other modules to access information of a smell object.

Implemented By: EcoOptimizer

Type of Module: Abstract Data Type

7.2.2 Base Refactorer Module (M2)

Secrets: How to handle file I/O operations for reading and writing code. How to parse source code into AST/CST representation and convert modified AST/CST back to source code. How to traverse, analyze and modify AST/CST trees.

Services: Offers an interface for other refactoring modules to implement.

Implemented By: EcoOptimizer

7.2.3 MakeStaticRefactorer Module (M3)

Secrets: How to identify method calls that don't use instance attributes, determine class hierarchies and inheritance relationships, transform instance method calls to static method calls and handle method annotations and type hints.

Services: Refactors the ***Member Ignoring Method (MIM)*** smell in a provided code file to improve energy efficiency.

Implemented By: EcoOptimizer

Type of Module: Abstract Object: A transformation component that modifies Python source code to optimize method calls.

Formalization of MakeStaticRefactorer

1. **Definitions**

   - *Source Code*: A Python program D consisting of a set of statements $S = \{s_1, s_2, \ldots, s_n\}$.
   - *Member Ignoring Method (MIM)*: A function f defined inside a class C that does not use instance attributes or methods.
   - *Static Method Transformation*: A function T that converts an instance method f into a static method f', where:

     $$T(f) = f' \text{ such that } \forall x \in C, f(x) = f'$$

   - *Refactored Code*: The transformed version of D, denoted as D', where all applicable f are replaced by f'.

2. **Transformation Process**

   - Parse D into an AST representation.
   - Identify functions f where:
     - $f \in C$ (i.e., f is a method within a class).
     - f does not reference any instance variables.
   - Modify f by:
     - Adding a '@staticmethod' decorator.
     - Removing the 'self' parameter from its signature.
   - Update all relevant method calls in D to reflect the static method transformation.

3. **Validation and Constraints**

   - Ensure correctness by preserving method functionality.
   - Verify that method calls in D remain semantically valid after transformation.
   - Maintain compatibility with subclasses and inherited methods.

7.2.4 UseListAccumulationRefactorer Module (M4)

Secrets: How to identify string concatenation operations within loops and find the scope of the concatenation. How to track variable assignments and reassignments across scopes. How to transform string concatenations into list accumulation patterns.

Services: Refactors the ***String Concatenation Inside Loop (SCL)*** smell in a provided code file to improve energy efficiency.

Implemented By: EcoOptimizer

7.2.5 UseAGeneratorRefactorer Module (M5)

Secrets: How to identify list comprehensions through operations. How to transform list comprehensions into generator expressions. How to preserve short-circuit evaluation behavior.

Services: Refactors the ***List Comprehension Instead of a Generator*** smell in a provided code file to improve energy efficiency.

Implemented By: EcoOptimizer

7.2.6 CacheRepeatedCallsRefactorer Module (M6)

Secrets: How to identify repeated function calls with same arguments, how to determine appropriate scope for caching. How to implement memoization patterns and handle side effects in cached functions.

Services: Refactors the ***Repeated Function Calls*** smell in a provided code file to improve energy efficiency and performance.

Implemented By: EcoOptimizer

7.2.7 Long Element Chain Module (M7)

Secrets: How to identify dictionary assignments, analyze the structure of nested dictionaries, and flatten them. How to identify all access calls associated with these dictionaries in the source code and determines how to update them to reflect the new flattened structure. How to maintain data structure integrity during transformation.

Services: Detects nested dictionaries in the source code using AST parsing, simplifies their structure by flattening them, and updates all associated access calls throughout the file. This improves code readability, reduces complexity, and ensures correctness while maintaining the program's intended behavior.

Implemented By: EcoOptimizer

Type of Module: Library: provides functionality for refactoring nested dictionary structures in Python code.

Long Element Chain Formalization

1. **Definitions**

   - *Nested Dictionary*: Dictionary $D = \{k_1 : \{k_2 : \ldots \{k_n : v\}\}\}$
   - *Flattened Dictionary*: $D' = \{k_1\_k_2\_\ldots\_k_n : v\}$
   - *Access Path*: Sequence $P = [k_1, k_2, \ldots, k_n]$

- *Composite Key*: String $c_k = \text{join}(P, \text{"\_"})$
- *Indentation*: Leading whitespace W from original code

2. **Refactoring Function**

   - Dictionary Flattening:

$$\begin{aligned} &\forall \text{ nested path } P \in D,\ \exists\ c_k \in D' \\ &D'[c_k] = D[P_1][P_2]\dots[P_n] \quad \text{(preserve } W\text{)} \end{aligned}$$

   - Access Path Update:

$$\begin{array}{lll} \text{Original Access:} & & D[P_1][P_2]\dots[P_n] \\ \text{Refactored Access:} & D'[c_k] & \text{where } c_k = \text{join}(P, \text{"\_"}) \end{array}$$

7.2.8 Long Parameter List Module (M8)

Secrets: How to identify functions or methods with long parameter lists, and encapsulate related parameters into objects or structures. How to group related parameters into objects/structures. How to update all call sites with new parameter structures. How to identify all function or method calls associated with these functions and updates their arguments to align with the refactored signature.

Services: Detects long parameter lists in functions or methods using AST parsing, simplifies their structure by grouping related parameters into objects or structures, and updates all associated function or method calls throughout the file. This improves code readability, reduces complexity, and ensures correctness while maintaining the program's intended behavior.

Implemented By: EcoOptimizer

7.2.9 Long Message Chain Refactorer (M9)

Secrets: How to analzye the syntax and structure of Python code to identify/classify long message chains, including both f-strings and non-f-string chains. How to extract method calls from identified long chains, and systematically break them into separate statements while preserving the original functionality and indentation. How to correct unmatched brackets during refactoring.

Services: Detects long message chains in Python source code and refactors them by splitting the chain into intermediate variables and a final result. This improves code readability and maintainability while retaining the original program behavior.

Implemented By: EcoOptimizer

Type of Module: Library: a reusable component that provides functionality for refactoring long message chains in Python code.

Long Message Chain Formalization

1. **Definitions**

   - *Input Line*: String L (e.g., `obj.a().b().c()`)
   - *Method Chain*: Sequence $M = [m_1, m_2, \ldots, m_n]$, where m_i is a method call
   - *Intermediate Variables*: Generated variables $V = [v_0, v_1, \ldots, v_{n-1}]$
   - *Indentation*: Leading whitespace W from L

2. **Refactoring Function**

   - For f-strings:

$$
\begin{aligned}
v_0 &= f_{\text{str}} \quad \text{(preserve } W\text{)} \\
v_1 &= v_0.m_1 \quad \text{(preserve } W\text{)} \\
&\vdots \\
\text{result} &= v_{n-1}.m_n \quad \text{(preserve } W\text{)}
\end{aligned}
$$

   - For regular chains:

$$
\begin{aligned}
v_0 &= m_1 \quad \text{(preserve } W\text{)} \\
v_1 &= v_0.m_2 \quad \text{(preserve } W\text{)} \\
&\vdots \\
\text{result} &= v_{n-1}.m_n \quad \text{(preserve } W\text{)}
\end{aligned}
$$

7.2.10 Long Lambda Function Refactorer (M10)

Secrets: How to analyze the syntax and structure of Python code to identify long lambda functions. How to locate lambda expressions, extract their arguments and bodies, and convert them into standard Python function definitions. How to properly handle nested structures (e.g., parentheses, brackets, and braces) within the lambda body, and truncates overly complex expressions at the first top-level comma for readability.

Services: Detects components of long lambda functions in Python source code and refactors them by converting the lambda expression into a standalone function with a unique name. This improves code readability, debugging, and maintainability. The module inserts the newly defined function in the appropriate scope, updates the original lambda usage with a function call, and validates the refactored code by maintaining its functionality and measuring energy efficiency improvements.

Implemented By: EcoOptimizer

Type of Module: Library: a reusable component that provides functionality for refactoring long lambda functions in Python code.

Long Lambda Function Formalization

1. **Definitions**

   - *Lambda Expression*: String $L =$ "lambda" A":" B where:
     - $A = [a_1, \ldots, a_n]$: Formal parameters
     - B : Body expression
   - *Truncated Body*: $B' = \begin{cases} B[0:i] & \text{if top-level comma at } i \\ B & \text{otherwise} \end{cases}$
   - *Function Name*: $F =$ "converted_lambda_" + line_num
   - *Indentation*:
     - W : Original line's whitespace
     - W_c : Parent block indentation
     - $D = 4$: Indentation delta (spaces)

2. **Transformation Rules**

   - **Function Generation**:

     $$\begin{aligned} \textsc{Definition}: &\quad W_c\text{"def "}F(A)\text{":"} \\ \textsc{Body}: &\quad (W_c + D)\text{"result = "}B' \\ \textsc{Return}: &\quad (W_c + D)\text{"return result"} \end{aligned}$$

   - **Lambda Replacement**:

     $$W\text{"lambda "}A\text{": "}B \quad \Rightarrow \quad WF$$

3. **Example**

   - *Before*:

     ```
     # Original lambda (line 42)
     result = map(lambda x: x**2, data)
     ```

   - *After*:

     ```
     def converted_lambda_42(x):     # W_c = ""
         result = x**2               # W_c + D (D=4)
         return result

     result = map(converted_lambda_42, data)
     ```

Key Relationships:

- F uniqueness via line_num
- $D = 4$ ensures PEP8 compliance
- W preservation maintains code structure

Key Relationships:

- F preserves lexical scope via W_c alignment
- B' ensures syntactic validity through comma truncation
- $\Delta = 4$ enforces PEP8 indentation rules
- W retention maintains original code structure

7.2.11 Plugin Initiator Module (M11)

Secrets: How to initialize the plugin, set up the environment for interaction, and handle configurations for plugin commands.

Services: Initializes and manages the plugin's setup including command registration, enabling its functionality without exposing the underlying initialization logic.

Implemented By: EcoOptimizer

7.2.12 Backend Communicator (M12)

Secrets: How to establish communication between the plugin and Source Code Optimizer, handle requests, and process responses.

Services: Provides an interface for sending requests to and receiving responses from Source Code Optimizer. Abstracts the communication details from modules within the plugin. Managers server status checks and handles backend communication errors.

Implemented By: EcoOptimizer

7.2.13 Smell Detector (M13)

Secrets: How to analyze Python code in active VS Code editor by accessing Source Code Optimizer endpoint.

Services: Detects code smells in Python scripts using Source Code Optimizer and provides structured data for further processing based on received output.

Implemented By: EcoOptimizer

Type of Module: Abstract Object: A component responsible for code smell detection and classification.

Smell Detection Formalization

1. **Definitions**

   - *Source Code*: A document D containing a sequence of statements $S = \{s_1, s_2, \ldots, s_n\}$.
   - *Code Smell*: A property C_k that signifies a potential issue in D, where:

     $$C_k = \{c_1, c_2, \ldots, c_m\}$$

     represents a set of detectable code smells.
   - *Detection Function*: A mapping $F : D \to C_k$, which evaluates D and returns a set of identified smells.
   - *Smell Report*: A structured output R summarizing detected smells and their locations in D.

2. **Detection Process**

   - Extract syntactic and semantic features from D.
   - Apply detection rules to identify instances of C_k.
   - Generate a structured smell report R containing:
     - Smell type c_i
     - Line number(s) $L = \{l_1, l_2, \ldots, l_p\}$
     - Severity level σ based on predefined metrics

3. **Classification and Output**

   - Structure R for further analysis by the refactoring module.
   - Ensure results are accessible to downstream processing components.

4. **Validation and Adaptation**

   - Maintain a feedback loop to refine detection accuracy.
   - Adapt detection parameters based on observed refactoring effectiveness.

7.2.14 File Highlighter Module (M14)

Secrets: How to identify code regions requiring attention and how to highlight those regions in the file.

Services: Highlights areas in files that contain code smells and can be refactored, assisting developers in identifying problem areas quickly.

Implemented By: EcoOptimizer

7.2.15 Hover Manager Module (M15)

Secrets: Understanding how to detect hover events over source code, retrieve relevant contextual information dynamically, and generate structured hover content. This includes identifying code smells at a given position, constructing a 'MarkdownString' representation, and embedding interactive refactoring commands.

Services: Manages hover interactions within the VSCode extension. When a user hovers over a detected code smell, it dynamically provides a 'MarkdownString' containing information about the smell and options to refactor it. The module supports both single-instance and type-wide refactoring actions.

Implemented By: EcoOptimizer

Type of Module: Library: a reusable component responsible for handling hover events and providing interactive refactoring commands within the extension.

Hover Interaction Formalization

1. **Definitions**

   - *Interaction Space*: A document D consisting of discrete positions $P = \{p_1, p_2, \ldots, p_n\}$ where detected smells can be associated.
   - *Smell Data*: A set $S = \{s_1, s_2, \ldots, s_m\}$, where each s_i is a detected smell with:
     - s_i.location — position range where the smell occurs
     - s_i.description — information about the smell
   - *Response Function*: A mapping $H : P \to S$ that associates positions in D with detected smells.

2. **Interaction Model**

$$H(p) = \begin{cases} s_i, & \text{if } p \text{ corresponds to a detected smell} \\ \varnothing, & \text{otherwise} \end{cases}$$

3. **Contextual Response Construction**

   - Given $p \in P$, retrieve $s_i \in S$.
   - Construct a structured response containing s_i.description and s_i.transformationOptions.
   - Present the response in an interactive format that allows further refinement or transformation.

7.2.16 Cache Manager Module (M20)

Secrets: How to persist and retrieve smell detection results, manage cache invalidation, handle cache state transitions and synchronize cache with file system changes.

Services: Provides persistent storage for smell detection results and the ability to retrieve cached smell data, invalidate cache entries for specific files, clear entire cache. Coordinates cache updates with file system events.

Implemented By: EcoOptimizer

7.2.17 Filter Manager Module (M21)

Secrets: How to handle and respond to filter criteria across sessions. How to handle filter combinations concurrently with UI synchronization.

Services: Provides functionality to set and get filter criteria for different smell types and apply filters to smell data. This module can clear all active filters, save filter state to persistent storage, restore filter state from storage and validate filter criteria.

Implemented By: EcoOptimizer

7.3 Software Decision Module

7.3.1 Measurements Module (M16)

Secrets: How to measure energy consumption and carbon emissions of a given Python program using the CodeCarbon library, including managing temporary directories for storing output, executing the program, and processing the emissions data from a CSV file.

Services: Provides functionality for measuring the energy consumption and carbon emissions of a provided code file. This module handles execution, tracking, and data extraction, ensuring that the emissions data is available for further analysis.

Implemented By: CodeCarbonEnergyMeter

7.3.2 Pylint Analyzer Module (M17)

Secrets: The internal design and execution of static code analysis using Pylint and AST parsing, including custom detection and structuring of smells. These details are hidden from external modules.

Services: The module provides the following services:

- Executes Pylint analysis on Python source code files.
- Performs AST-based custom checks for specific code smells.

- Filters and structures the analysis results into a standardized format for further processing.

Implemented By: `EcoOptimizer`

7.3.3 Smell Refactorer Module (M18)

Secrets: How to utilize Source Code Optimizer endpoint for refactoring specific code smells, and how to sanitize received output.

Services: Refactors code smells in Python scripts using Source Code Optimizer and determines if received output is valid format.

Implemented By: EcoOptimizer

7.3.4 Refactor Manager Module (M19)

Secrets: How to coordinate the sequence of refactorings, apply transformations to source code while maintaining correctness, validate the impact of each step, and enable reversion when necessary. Ensures that refactoring operations preserve program behavior.

Services: Manages the execution of refactorings, determines whether modifications should be committed to the source file, and allows users to revert changes if needed.

Implemented By: EcoOptimizer

Type of Module: Abstract Object: A conceptual entity that organizes and controls the refactoring process.

Refactoring Execution Formalization

1. **Definitions**
   - *Source Code*: A document D consisting of a sequence of statements $S = \{s_1, s_2, \ldots, s_n\}$.
   - *Code Smell*: An identified issue σ in D that may impact maintainability, readability, or performance.
   - *Refactoring Rule*: A transformation function $R : D \to D'$ that modifies S while improving energy consumption.
   - *Refactoring Sequence*: An ordered set of transformations $\mathcal{R} = [R_1, R_2, \ldots, R_m]$ applied sequentially to D.
2. **Refactoring Process**
   - Given a detected smell σ, identify the corresponding transformation R_σ.
   - Apply R_σ to D, producing D'.

- Track all modifications to ensure consistency and correctness.

3. **Modification Tracking**
   - Maintain a count of applied transformations per smell type:

$$C_\sigma = C_\sigma + 1$$

   - Store modified versions of D to enable rollback.
   - Generate output files when refactoring is applied.

4. **Validation and Reversion**
   - After applying R_σ, verify correctness:

$$V(D') \rightarrow \{\text{valid}, \text{invalid}\}$$

   - If invalid, restore the previous state D.
   - Allow user-driven reversion to the original state if necessary.

7.3.5 Energy Metrics Module (M22)

Secrets: How to collect and aggregate code metrics from backend output. How to handle metric collection errors and edge cases. How to manage metric data consistency across sessions

Services: Collects code metrics for specified files and outputs metrics data to suitable formats. Provides functionality to display metrics in the VS Code UI and save metrics data to persistent storage.

Implemented By: EcoOptimizer

7.3.6 View Provider Module (M23)

Secrets: How to implement IDE specific views for different data types. How to manage view state and updates efficiently across multiple views. How to handle dynamic content updates based on user interactions. How to coordinate between different views (smells, metrics, filters, refactoring)

Services: Provides view display of detected smells with hierarchical format. Provides energy metrics data visualization and updates. Provides filter selection and management interface. Provides refactoring details and preview display. Provides methods to refresh and update all views.

Implemented By: EcoOptimizer

7.3.7 Event Manager Module (M24)

Secrets: How to implement IDE event handling system. How to manage event propagation and subscription lifecycle. How to handle concurrent event processing and state updates and coordinate between different event types.

Services: Provides event subscription management for workspace changes, file system modifications and UI updates. Provides functionality to register and unregister event listeners, handle event propagation, manage event state and cleanup.

Implemented By: EcoOptimizer

8 Traceability Matrix

This section shows two traceability matrices: between the modules and the requirements and between the modules and the anticipated changes.

Req.	Modules
FR1	M2, M3, M4, M5, M6, M7, M10, M8, M9, M16, M17, M18, M19, M22, M13
FR2	M1, M17, M13
FR3	M19, M3, M4, M5, M6, M7, M10, M8, M9, M16, M18
FR4	M3, M4, M5, M6, M7, M10, M8, M9, M19
FR5	M3, M4, M5, M6, M7, M10, M8, M9, M16, M17
FR6	M23, M22, M16
FR7	M2, M3, M4, M5, M6, M7, M10, M8, M9, M16, M17, M23, M24
FR8	M16, M17, M23 M22
FR9	M21, M23
FR10	M14, M11, M12, M13, M18, M15
FR11	M24, M11, M14, M23
FR12	M2, M3, M4, M5, M6, M7, M10, M8, M9, M15, M12, M14, M18
FR13	M21, M24, M11
FR14	M24, M2, M3, M4, M5, M6, M7, M10, M8, M9, M16, M17, M18, M19, M22, M13
FR15	M23, M22, M24
FR16	M24, M13, M11
FR17	M20

Table 2: Trace Between Functional Requirements and Modules

Req.	Modules
LFR-AP 1	M11, M12, M18, M23, M22
LFR-AP 2	M11, M23, M22 M15, M14
LFR-ST 1	M11, M14, M15, M23

Table 3: Trace Between Look & Feel Requirements and Modules

Req.	Modules
UHR-EOU 1-2	M11, M14, M15, M23, M21
UHR-PS1 1	M??, M23, M11
UHR-LRN 1	M11, M12, M13, M18, M14, M15, M19
UHR-LRN 2	Not implemented in code
UHR-UPL 1	M11, M12, M13, M18, M14, M15, M19
UHR-ACS 1	M11, M12, M13, M18, M14, M15, M19, M23

Table 4: Trace Between Usability & Humanity Requirements and Modules

Req.	Modules
PR-SL 1	M17
PR-SL 2	M2, M3, M4, M5, M6, M7, M10, M8, M9
PR-CR 1	M2, M3, M4, M5, M6, M7, M10, M8, M9, M16, M17
PR-PAR 2	M17
PR-PAR 3	M2, M3, M4, M5, M6, M7, M10, M8, M9
PR-RFT 1	M2, M3, M4, M5, M6, M7, M10, M8, M9, M16, M17
PR-RFT 2	M2, M3, M4, M5, M6, M7, M10, M8, M9
PR-SER 1	M2
PR-LR 1	M1, M2, M3, M4, M5, M6, M7, M10, M8, M9, M16, M17

Table 5: Trace Between Performance Requirements and Modules

Req.	Modules
OER-EP 1	N/A
OER-EP 2	N/A
OER-WE 1	M16
OER-IAS 1	To be removed
OER-IAS 2	M11, M12, M13, M18, M14, M15, M19
OER-IAS 3	To be removed
OER-PR 1	M1, M2, M3, M4, M5, M6, M7, M10, M8, M9, M16, M17
OER-RL 1	M1, M2, M3, M4, M5, M6, M7, M10, M8, M9, M16, M17
OER-RL 2	N/A

Table 6: Trace Between Operational & Environmental Requirements and Modules

Req.	Modules
MS-MNT 1	M2
MS-MNT 2	Not implemented in code
MS-MNT 3	None
MS-MNT 4	N/A
MS-MNT 5	M1, M2, M3, M4, M5, M6, M7, M10, M8, M9, M16, M17
MS-SP 1	Not implemented in code

Table 7: Trace Between Maintenance & Support Requirements and Modules

Req.	Modules
SR-AR 1	To be removed
SR-AR 2	M16
SR-IR 1	M2, M3, M4, M5, M6, M7, M10, M8, M9
SR-PR 1	N/A
SR-PR 2	M2, M3, M4, M5, M6, M7, M10, M8, M9, M16, M17
SR-AUR 1	M2, M3, M4, M5, M6, M7, M10, M8, M9, M16, M17
SR-AUR 2	M2, M3, M4, M5, M6, M7, M10, M8, M9, M16, M17
SR-IM 1	N/A

Table 8: Trace Between Security Requirements and Modules

Req.	Modules
CULT 1-3	M11, M12, M13, M18, M14, M15, M19
CL-LR 1	M2, M3, M4, M5, M6, M7, M10, M8, M9, M16, M17,
CL-SCR 1	M2, M3, M4, M5, M6, M7, M10, M8, M9, M16, M17,

Table 9: Trace Between Cultural and Compliance Requirements and Modules

AC	Modules
AC1	M11, M14 , M15, M19
AC2	M11 , M18, M19
AC3	M2, M3, M4, M5, M6, M7, M8
AC4	M13, M17
AC5	M16, M17, M13

Table 10: Trace Between Anticipated Changes and Modules

9 Use Hierarchy Between Modules

In this section, the uses hierarchy between modules is provided. Parnas (1978) said of two programs A and B that A *uses* B if correct execution of B may be necessary for A to complete the task described in its specification. That is, A *uses* B if there exist situations in which the correct functioning of A depends upon the availability of a correct implementation of B. Figure 1 illustrates the use relation between the modules. It can be seen that the graph

is a directed acyclic graph (DAG). Each level of the hierarchy offers a testable and usable subset of the system, and modules in the higher level of the hierarchy are essentially simpler because they use modules from the lower levels. Furthermore, figure 2 illustrates the use relation between the modules of the VS Code Plugin, vis-a-vis the Source Code Optimizer displayed in figure 1. In this figure for the plugin, the modules at the lowest level are the ones closest to the user, and going up in the module indicates moving towards the backend of the project, which is the Source Code Optimizer. It is a layered architecture where each module serves a separate concern.

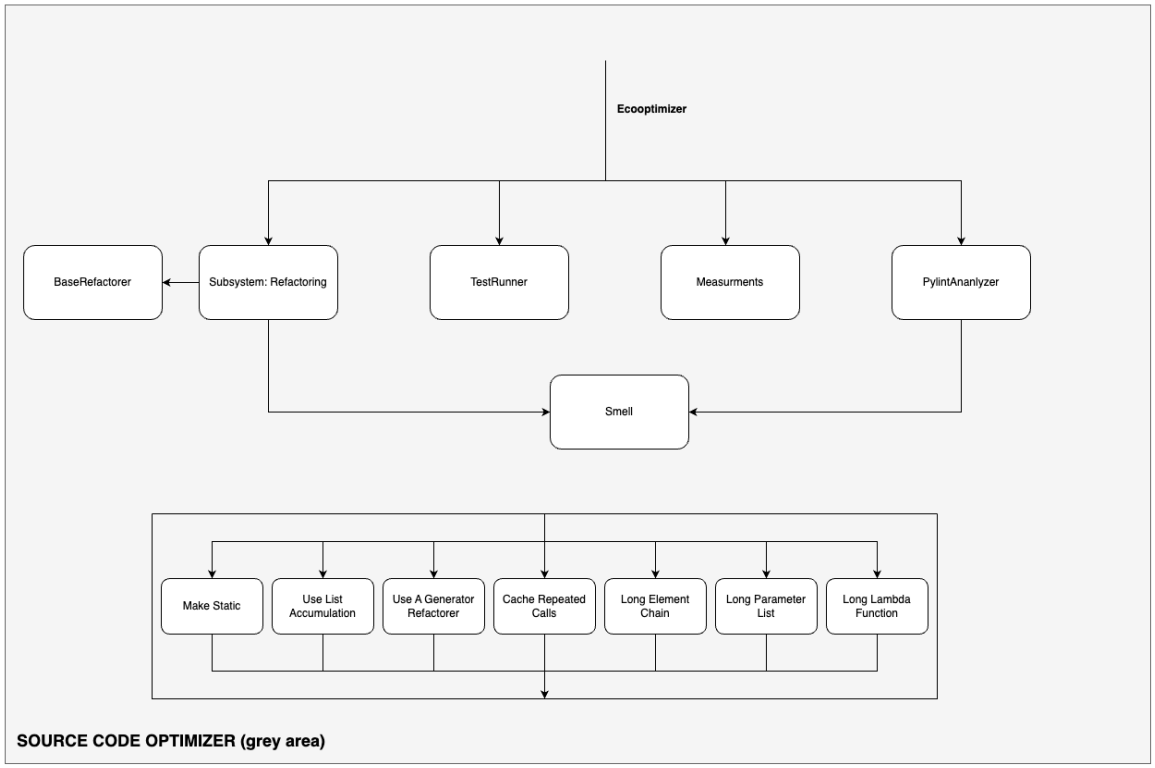

Figure 1: Use hierarchy among modules

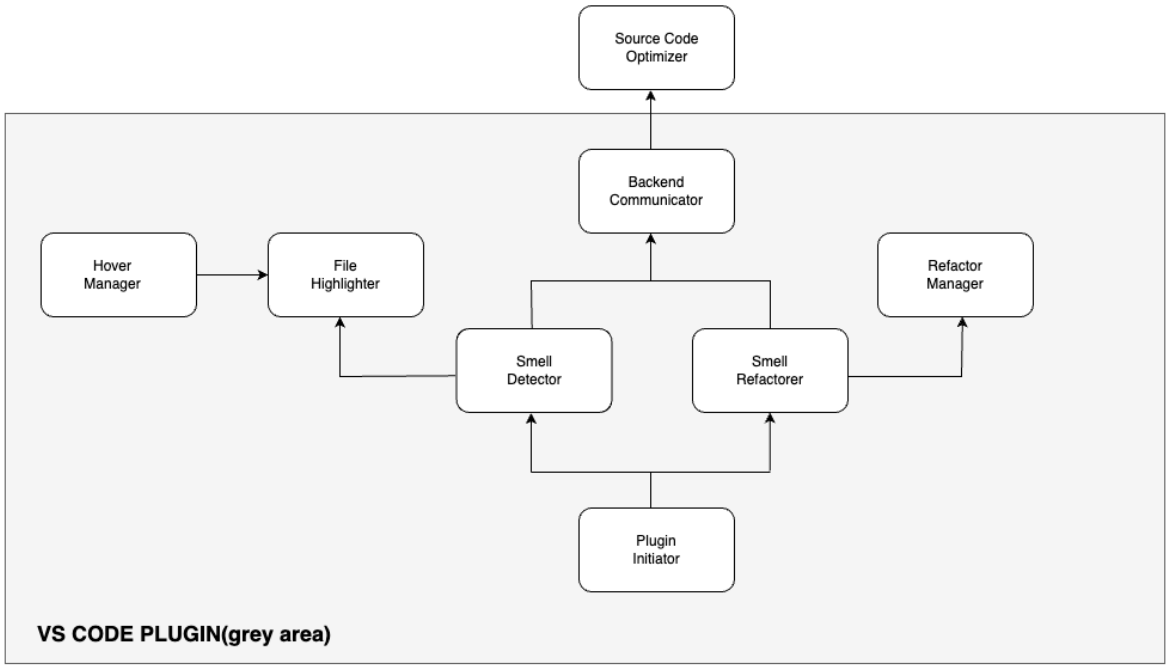

Figure 2: Use hierarchy among modules

10 User Interfaces

The project is exposed to the user through the VS Code Plugin, which can be installed in any developer's local VS Code Editor. Figure 3 highlights the intial interface that the user accesses when installating and enabling the plugin. Commands of the plugin are available through various routes, one of them is the Command Palette as show in figure 4.

Commands are searchable and can be applied. Figure 5 showcases the application of the "Eco: Refactor Plugin: Detect Smells" command, which at the end highlights all lines containing code smells with yellow lines. On hover, information is available about the specific code smell. The user also has the option to select or place their cursor on a specific line that they want to refactor, and run the "Eco: Refactor Plugin: Refactor Smell" command in order to refactor that specific line of code. Figure 6 provides a visual of this command in progress, with updates being provided in bottom right corner.

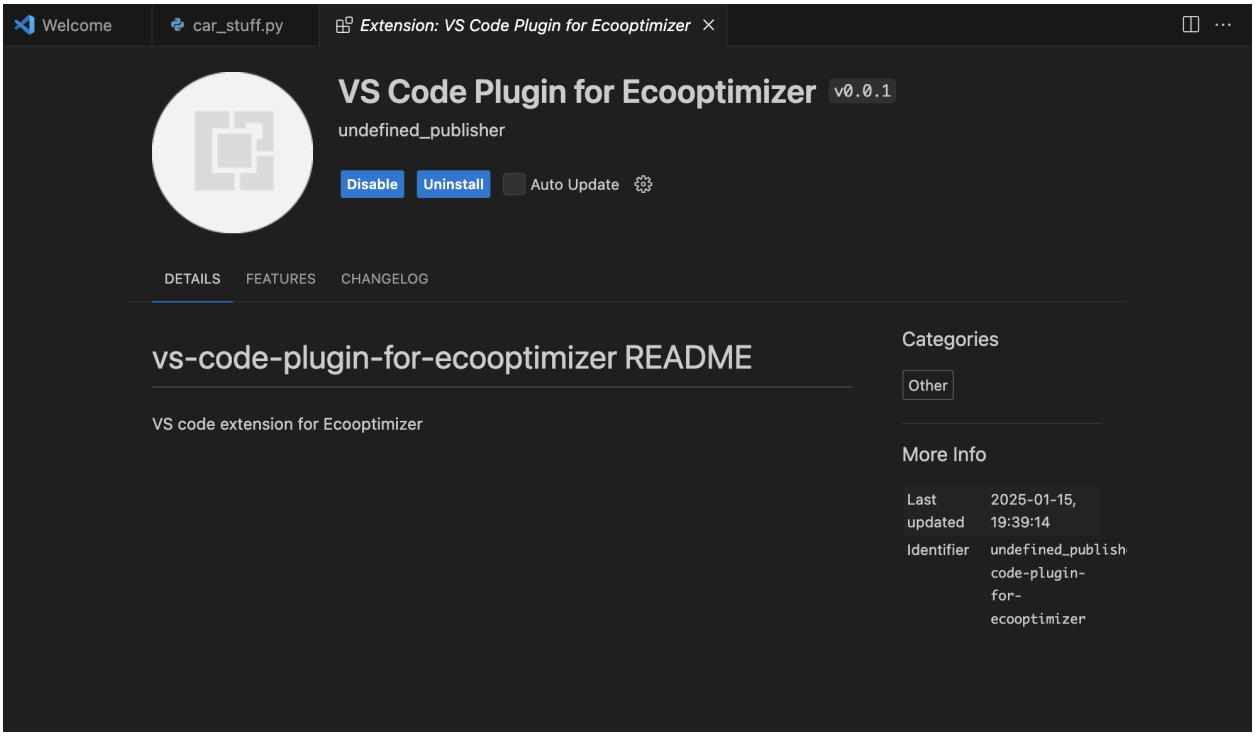

Figure 3: VS Code Plugin Setup

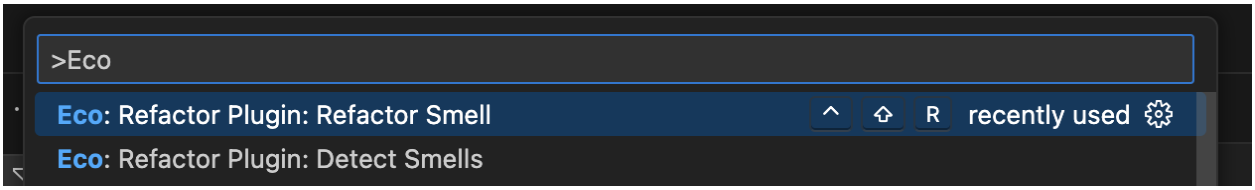

Figure 4: VS Code Plugin Commands

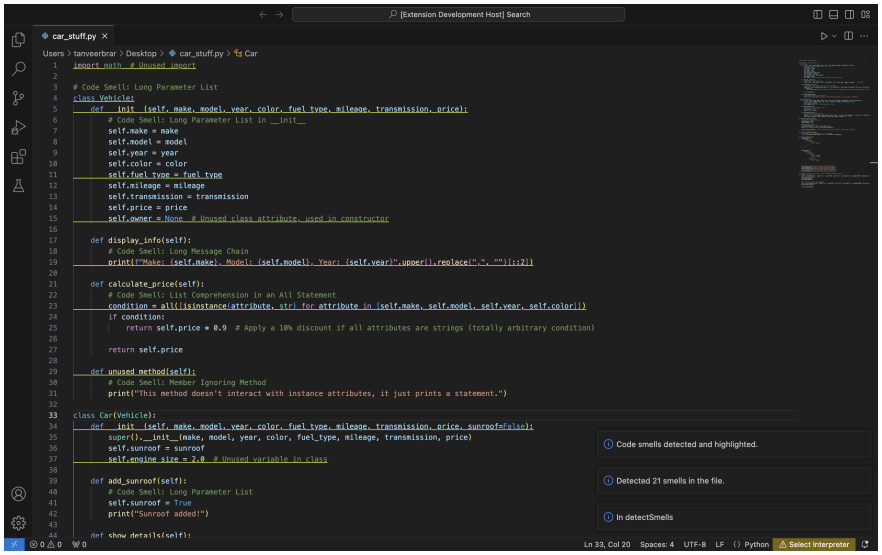

Figure 5: VS Code Code Analysis Interaction

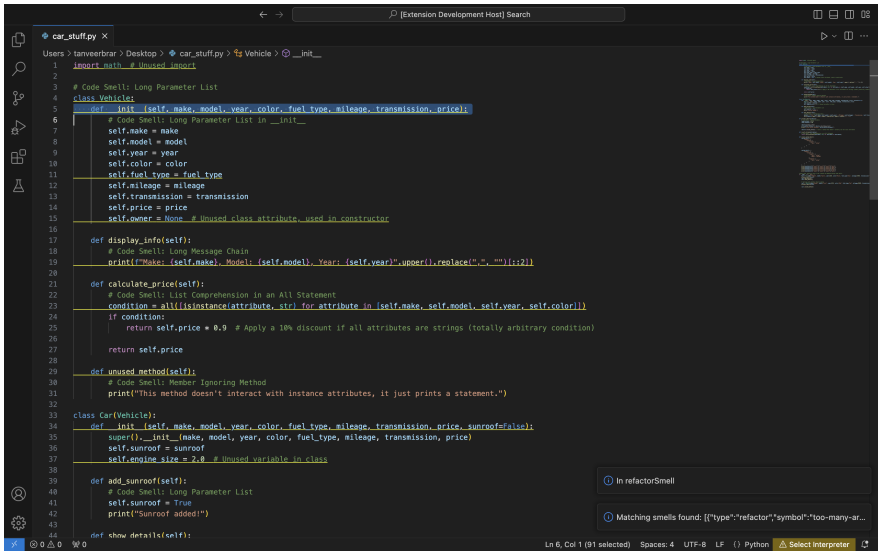

Figure 6: VS Code Code Refactoring Interaction(in progress for selected line)

11 Design of Communication Protocols

12 Timeline

All code and corresponding documentation was aimed to be completed before Jan 6th our Demo date for our supervisor.

Table 11: Timeline

Module Name	**Team Member**	**Due Date**
Base Refactorer	Sevhena Walker	Jan 6, 2025
Complex List Comprehension	Nivetha Kuruparan	Jan 6, 2025
Long Element Chain	Ayushi Amin	Jan 6, 2025
Long Lambda Function	Mya Hussain	Jan 6, 2025
Long Message Chain	Mya Hussain	Jan 6, 2025
Member Ignoring Method	Sevhena Walker	Jan 6, 2025
Repeated Calls	Nivetha Kuruparan	Jan 6, 2025
String Concatenation in Loop	Sevhena Walker	Jan 6, 2025
Long Parameter List	Tanveer Brar	Jan 6, 2025
Smell	All	Jan 31, 2025
Analyzers	All	Jan 31, 2025
Measurements	All	Jan 31, 2025

References

David L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15(2):1053–1058, December 1972.

David L. Parnas. Designing software for ease of extension and contraction. In *ICSE '78: Proceedings of the 3rd international conference on Software engineering*, pages 264–277, Piscataway, NJ, USA, 1978. IEEE Press. ISBN none.

D.L. Parnas, P.C. Clement, and D. M. Weiss. The modular structure of complex systems. In *International Conference on Software Engineering*, pages 408–419, 1984.