Module Interface Specification for Software Engineering

Team 4, EcoOptimizers

Nivetha Kuruparan
Sevhena Walker
Tanveer Brar
Mya Hussain
Ayushi Amin

April 5, 2025

1 Revision History

Date	Name	Notes
January 17th, 2025	All	Initial Draft
March 24th, 2025	Mya Hussain	Removed Pythonic Syntax Mentions
March 24th, 2025	Mya Hussain	Added Some References to Local Functions
March 24th, 2025	Mya Hussain	Modified Wrong Environment Variables
March 25th, 2025	Tanveer Brar	Added MIS for new plugin modules
March 28th, 2025	Tanveer Brar	Addressed TA and Peer Review Feedback Issues

2 Symbols, Abbreviations and Acronyms

See SRS Documentation.

Contents

3 Introduction

The following document details the Module Interface Specifications (MIS) for the Source Code Optimizer project. The Source Code Optimizer is a software tool designed to analyse, refactor, and optimise Python source code to improve energy efficiency, maintainability, and performance. This tool incorporates a combination of static code analysis using Pylint, abstract syntax tree (AST) parsing, and custom refactoring techniques to detect and address various code smells in Python programs.

The application allows developers to identify inefficient coding patterns, refactor them into optimized alternatives, and validate the results through built-in testing mechanisms. Key features include support for custom smell detection, energy profiling, and modular refactorers tailored to specific code smells, such as long method chains or inefficient list comprehensions. By automating parts of the optimization process, the Source Code Optimizer helps developers have the option of choosing to reduce emissions and produce more efficient software.

Complementary documents include the System Requirement Specifications (SRS) and Module Guide (MG). The full documentation and implementation can be found at: https://github.com/ssm-lab/capstone--source-code-optimizer

4 Notation

The following table summarizes the primitive data types used by Software Engineering.
The specification of Software Engineering uses some derived data types: sequences, strings, and tuples. Sequences are lists filled with elements of the same data type. Strings are sequences of characters. Tuples contain a list of values, potentially of different types. In addition, Software Engineering uses functions, which are defined by the data types of their inputs and outputs. Local functions are described by giving their type signature followed by their specification.

Data Type	**Notation**	**Description**
optional	?	denotes a variable as optional
any type	Any	any data type is acceptable
character	char	a single symbol or digit
String	str	a sequence of characters
integer	$\mathbb{Z}$	a number without a fractional component in (-∞, ∞)
natural number	$\mathbb{N}$	a number without a fractional component in [1, ∞)
real	$\mathbb{R}$	any number in (-∞, ∞)
boolean	$\mathbb{B}$	True or False
code smell	Smell	a collection of data representing a code smell
path	Path	Data object representing a path in a filesystem
list	list[T]	an ordered collection of objects of type T
set	set[T]	an unordered collection of *unique* objects of type T
dictionary	dict[key] = value	data structure containing multiple key-value pairs
AST Node	AST	AST node representing any AST node
AST Constant	Constant	AST node representing a constant
AST Function Definition	FuncDef	AST node representing a function definition
AST Module	Module	AST node representing a Module
AST Class Definition	ClassDef	ast node representing a class definition
AST Call	Call	ast node representing a function call
AST Lambda	Lambda	ast node representing a lambda function
AST List Comprehension	ListComp	ast node representing a list comprehension
AST Generator Expression	GenExp	ast node representing a generator expression
current instance	self	a reference to the current instance of a module

Table 1: MIS Notation

5 Module Decomposition

The following table is taken directly from the Module Guide document for this project.

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 Testing Functionality Module SmellRefactorer Module RefactorManager Module EnergyMetrics Module ViewProvider Module EventManager Module

Table 2: Module Hierarchy

6 MIS of Smell Data Type

`Smell`

6.1 Module

Contains data related to a code smell.

6.2 Uses

None

6.3 Syntax

Exported Constants: None
Exported Access Programs: None

6.4 Semantics

6.4.1 State Variables

- `absolutePath: str`: Absolute path to the source file containing the smell.
- `column: int`: Starting column in the source file where the smell is detected.
- `confidence: str`: Confidence level for the smell detection.
- `endColumn?: int`: Ending column for the smell location, if applicable.
- `endLine?: int`: Ending line number for the smell location, if applicable.
- `occurences: dict`: Contains positional data related to where the smell is located in a code file.
- `message: str`: Descriptive message explaining the smell.
- `messageId: str`: Unique identifier for the specific message or warning.
- `module: str`: Module or component name containing the smell.
- `obj: str`: Specific object associated with the smell.
- `path: str`: Relative path to the source file from the project root.
- `symbol: str`: Symbol or code construct involved in the smell.
- `type: str`: Type or category of the smell.

6.4.2 Environment Variables

None

6.4.3 Assumptions

- All values provided to the fields of `Smell` conform to the expected data types and constraints.

6.4.4 Access Routine Semantics

`Smell()`

- **transition**: Creates a dictionary-like structure with the defined attributes representing a code smell.
- **output**: Returns a `Smell` instance.

6.4.5 Local Functions

None.

7 MIS of BaseRefactorer

`BaseRefactorer`

7.1 Module

The base interface that all refactorers inherit from, providing common functionality for file I/O, AST manipulation, code validation, and energy measurement integration.

7.2 Uses

None

7.3 Syntax

Exported Constants: None
Exported Access Programs:

Name	In	Out	Exceptions
`BaseRefactorer`	`output_dir: Path`	`self`	None
`refactor`	`file_path: Path, smell: Smell, initial_emissions:` $\mathbb{R}$	None	`IOError`
`parse_ast`	`source: str`	`AST`	`SyntaxError`
`validate_transformation`	`original: AST, modified: AST`	`bool`	None
`write_output`	`path: Path, content: str`	None	`IOError`
`measure_energy`	`before: float, after: float`	`float`	None
`apply_transformation`	`node: AST`	`AST`	None

7.4 Semantics

7.4.1 State Variables

- `output_dir: Path`: Directory for output files
- `temp_dir: Path`: Directory for temporary files during processing
- `ast_cache`: Cache for parsed AST trees
- `transformation_state`: Tracks current refactoring state
- `energy_metrics`: Stores energy consumption measurements

7.4.2 Environment Variables

- `WORKSPACE_ROOT`: Root directory of the workspace
- `FILE_PERMISSIONS`: Required file access permissions
- `TEMP_DIR_CONFIG`: Configuration for temporary directory

7.4.3 Assumptions

- Input files are valid Python scripts
- Output directory exists or can be created
- Write permissions are available for output directory
- AST transformations preserve program semantics

7.4.4 Access Routine Semantics

`BaseRefactorer(output_dir: Path)`

- **Transition**:
  - Initializes temporary directory using `_create_temp_dir`
  - Sets up AST cache using `_setup_ast_cache`
  - Configures transformation state
  - Initializes energy metrics
- **Output**: `self`
- **Exception**: None

`refactor(file_path: Path, smell: Smell, initial_emissions:` $\mathbb{R}$`)`

- **Transition**:
  - Reads and parses source file using `parse_ast`
  - Applies transformation using `apply_transformation`
  - Validates changes using `validate_transformation`
  - Measures energy impact using `measure_energy`
  - Writes output using `write_output` if valid
  - Cleans up using `_cleanup_temp_files`
- **Output**: None
- **Exception**: `IOError` if file operations fail

`parse_ast(source: str)`

- **Transition**: Parses source code into AST representation
- **Output**: AST object
- **Exception**: `SyntaxError` for invalid Python code

`validate_transformation(original: AST, modified: AST)`

- **Transition**: Verifies semantic equivalence between ASTs
- **Output**: Boolean indicating valid transformation
- **Exception**: None

`write_output(path: Path, content: str)`

- **Transition**: Writes transformed code after `_validate_permissions`
- **Output**: None
- **Exception**: `IOError` for file system issues

`measure_energy(before: float, after: float)`

- **Transition**: Calculates energy consumption difference
- **Output**: Float representing energy impact
- **Exception**: None

`apply_transformation(node: AST)`

- **Transition**: Abstract method for specific refactoring logic
- **Output**: Modified AST
- **Exception**: None

7.4.5 Local Functions

`_create_temp_dir()`

- **Transition**: Creates a temporary directory for storing intermediate files
- **Output**: Path to created directory
- **Exception**: `IOError` if directory creation fails

`_setup_ast_cache()`

- **Transition**: Initializes cache for storing parsed AST nodes
- **Output**: None
- **Exception**: None

`_cleanup_temp_files()`

- **Transition**: Removes all temporary files and directories
- **Output**: None
- **Exception**: `IOError` if cleanup fails

`_validate_permissions()`

- **Transition**: Verifies read/write permissions for target files
- **Output**: Boolean indicating if permissions are valid
- **Exception**: None

`_update_energy_metrics()`

- **Transition**: Updates energy consumption measurements
- **Output**: None
- **Exception**: None

8 MIS of LongMessageChainRefactorer

`LongMessageChainRefactorer`

8.1 Module

LongMessageChainRefactorer is a module that identifies and refactors long message chains in Python code to improve readability, maintainability, and performance. It specifically handles long chains by breaking them into separate statements, ensuring proper refactoring while maintaining the original functionality.

8.2 Uses

- Uses `Smell` interface for data access
- Inherits from `BaseRefactorer`

8.3 Syntax

8.3.1 Exported Constants

None

8.3.2 Exported Access Programs

Name	In	Out	Exceptions
`apply_transformation`	`node: AST`	`AST`	None
`identify_chains`	`node: AST`	`List[Chain]`	None
`extract_methods`	`chain: Chain`	`List[str]`	None
`generate_vars`	`methods: List[str]`	`List[str]`	None

8.4 Semantics

8.4.1 State Variables

- `chain_patterns`: Dictionary mapping chain types to their patterns
- `intermediate_vars`: List of generated variable names
- `indentation_map`: Mapping of line numbers to indentation levels

8.4.2 Environment Variables

Inherits from `BaseRefactorer`

8.4.3 Assumptions

- Message chains are properly terminated
- Variable names generated do not conflict with existing ones
- Indentation is consistent within code blocks

8.4.4 Access Routine Semantics

`apply_transformation(node: AST)`

- **Transition**:
  - Identifies message chains using `identify_chains`
  - Extracts methods using `extract_methods`
  - Generates intermediate variables using `generate_vars`
  - Validates chain breaks using `_validate_chain_break`
  - Preserves code formatting using `_preserve_indentation`
  - Reconstructs code with intermediate assignments
- **Output**: Modified AST
- **Exception**: None

`identify_chains(node: AST)`

- **Transition**: Analyzes AST for message chain patterns using `_analyze_chain_complexity`
- **Output**: List of identified chains
- **Exception**: None

`extract_methods(chain: Chain)`

- **Transition**: Extracts individual method calls from chain using `_preserve_indentation`
- **Output**: List of method call strings
- **Exception**: None

`generate_vars(methods: List[str])`

- **Transition**: Creates unique variable names using `_generate_unique_name`
- **Output**: List of variable names
- **Exception**: None

8.4.5 Local Functions

`_analyze_chain_complexity(chain: Chain)`

- **Transition**: Analyzes the length and complexity of method call chains
- **Output**: Integer representing chain complexity
- **Exception**: None

`_preserve_indentation(line: str)`

- **Transition**: Extracts and preserves the indentation level of code lines
- **Output**: String containing the indentation spaces
- **Exception**: None

`_validate_chain_break(original: str, parts: List[str])`

- **Transition**: Verifies that breaking the chain preserves functionality
- **Output**: Boolean indicating if chain break is valid
- **Exception**: None

`_generate_unique_name(base: str)`

- **Transition**: Creates a unique variable name based on the base string
- **Output**: String containing unique variable name
- **Exception**: None

9 MIS of LongLambdaFunctionRefactorer

`LongLambdaFunctionRefactorer`

9.1 Module

LongLambdaFunctionRefactorer is a module that refactors long lambda functions in Python code by converting them into normal functions. This improves code readability, maintainability, and performance, while reducing potential energy consumption.

9.2 Uses

`BaseRefactorer`

9.3 Syntax

9.3.1 Exported Constants

None

9.3.2 Exported Access Programs

Name	In	Out	Exceptions
`LongLambdaFunctionRefactorer`	`output_dir: Path`	`self`	None
`refactor`	`file_path: Path, smell: Smell, initial_emissions:` $\mathbb{R}$	None	`IOError`
`identify_lambdas`	`node: AST`	`List[Lambda]`	None
`extract_lambda_components`	`lambda_node: Lambda`	`(List[str], str)`	None
`generate_function_name`	`lambda_body: str`	`str`	None

9.4 Semantics

9.4.1 State Variables

- **lambda_registry**: Dictionary mapping lambda nodes to their locations
- **function_counter**: Counter for generating unique function names
- **scope_stack**: Stack tracking current scope for function placement

9.4.2 Environment Variables

Inherits from `BaseRefactorer`

9.4.3 Assumptions

- Input code contains valid Python syntax
- Lambda functions are properly defined
- Generated function names do not conflict with existing ones
- Scope information is correctly maintained

9.4.4 Access Routine Semantics

`LongLambdaFunctionRefactorer(output_dir: Path)`

- **Transition**:
  - Initializes base refactorer
  - Sets up lambda registry
  - Initializes function counter
  - Configures scope tracking
- **Output**: `self`
- **Exception**: None

`refactor(file_path: Path, smell: Smell, initial_emissions:` $\mathbb{R}$`)`

- **Transition**:
  - Reads source file
  - Identifies long lambda functions
  - Converts lambdas to named functions
  - Validates transformation
  - Writes refactored code if valid
- **Output**: None
- **Exception**: `IOError` if file operations fail

`identify_lambdas(node: AST)`

- **Transition**: Analyzes AST node for lambda function definitions
- **Output**: List of identified lambda nodes
- **Exception**: None

`extract_lambda_components(lambda_node: Lambda)`

- **Transition**: Extracts parameters and body from lambda function
- **Output**: Tuple of parameter list and body string
- **Exception**: None

`generate_function_name(lambda_body: str)`

- **Transition**: Creates descriptive name based on lambda body
- **Output**: Generated function name
- **Exception**: None

9.4.5 Local Functions

`_analyze_lambda_complexity(node: Lambda)`

- **Transition**: Analyzes lambda function complexity based on length and nesting
- **Output**: Integer representing complexity score
- **Exception**: None

`_find_insertion_point(node: AST)`

- **Transition**: Determines optimal location for new function definition
- **Output**: AST node indicating insertion point
- **Exception**: None

`_validate_scope(node: AST)`

- **Transition**: Verifies scope compatibility for function placement
- **Output**: Boolean indicating valid scope
- **Exception**: None

`_create_function_def(name: str, params: List[str], body: str)`

- **Transition**: Generates AST node for new function definition
- **Output**: FunctionDef AST node
- **Exception**: None

10 MIS of LongParameterListRefactorer

`LongParameterListRefactorer`

10.1 Module

LongParameterListRefactorer is a module that identifies and refactors functions or methods with long parameter lists(detected beyond configured threshold) in Python code. The refactoring aims to improve code readability, maintainability, and energy efficiency by encapsulating related parameters into objects and removing unused ones.

10.2 Uses

- Uses `Smell` interface for data access
- Inherits from `BaseRefactorer`
- Inherits from Python's `ast` module's `NodeTransformer`

10.3 Syntax

10.3.1 Exported Constants

None

10.3.2 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
`LongParameterListRefactorer`	`output_dir: Path`	`self`	None
`refactor`	`file_path: Path, pylint_smell: Smell, initial_emissions:` $\mathbb{R}$	None	`TypeError`, `IOError`

10.4 Semantics

10.4.1 State Variables

None

10.4.2 Environment Variables

None

10.4.3 Assumptions

- Input files are valid Python scripts.
- Smells identified by `pylint_smell` include valid line numbers.
- Refactored code must pass the provided test suite.

10.4.4 Access Routine Semantics

`LongParameterListRefactorer(output_dir: Path)`

- **Transition**: Initializes the refactorer with the specified output directory.
- **Output**: `self`.
- **Exception**: None.

`refactor(file_path: Path, pylint_smell: Smell, initial_emissions:` $\mathbb{R}$`)`

- **Transition**:
  1. Reads the file at `file_path` and locates the target function using `pylint_smell`.
  2. Analyzes function body using `get_used_parameters` to identify used parameters and remove unused ones.
  3. For remaining parameters that exceed the configured limit:
     - Groups parameters using `classify_parameters`
     - Creates parameter classes using `create_parameter_object_class`
     - Updates function signature using `update_function_signature`
     - Updates parameter usages using `update_parameter_usages`
     - Updates function calls using `update_function_calls`
  4. Writes the refactored code to a temporary file.
- **Output**: None. Refactored file is saved if improvements are validated.
- **Exception**: Raises `IOError` if input file cannot be read. Raises `TypeError` if source file cannot be parsed into an AST.

10.4.5 Local Functions

1. `get_used_parameters(function_node: ast.FunctionDef, params: list[str]) -> list[str]`:
   - **transition**: Identifies parameters used within the function body.
   - **output**: List of names of parameters that are actually used in the function.
   - **exception**: None
2. `get_parameters_with_default_value(params: list[ast.Param]) -> key-value pairs`:
   - **transition**: Maps parameter names to their default values .
   - **output**: Key-value pairs mapping parameter names to their default values.
   - **exception**: None
3. `classify_parameters(params: list[str]) -> key-value pairs`:
   - **transition**: Classifies parameters into `data` and `config` groups based on naming conventions.
   - **output**: Key-value pairs with keys "data_params" and "config_params" mapping to lists of parameter names.
   - **exception**: None
4. `create_parameter_object_class(param_names: list[str], default_value_params: key-value pairs, class_name: str) -> ast.ClassDef`:
   - **transition**: Creates an AST class definition for encapsulating related parameters.
   - **output**: AST ClassDef node representing the parameter object class.
   - **exception**: None
5. `update_function_signature(function_node: ast.FunctionDef, classified_params: key-value pairs) -> ast.FunctionDef`:
   - **transition**: Updates function signatures to use encapsulated parameter objects.
   - **output**: Updated AST FunctionDef node with new parameter structure.
   - **exception**: None
6. `update_parameter_usages(function_node: ast.FunctionDef, classified_params: key-value pairs) -> ast.FunctionDef`:
   - **transition**: Updates parameter references in function body to use encapsulated object attributes.
   - **output**: Updated AST FunctionDef node with transformed parameter usages.

- **exception**: None

7. `update_function_calls(tree: ast.Module, function_node: ast.FunctionDef, used_params: list[str], classified_params: key-value pairs, classified_param_names: tuple[str, str], enclosing_class_name: str) -> ast.Module`:
   - **transition**: Updates all calls to the refactored function to use new parameter structure.
   - **output**: Updated AST Module node with transformed function calls.
   - **exception**: None

11 MIS of UseListAccumulationRefactorer

`UseListAccumulationRefactorer`

11.1 Module

The `UseListAccumulationRefactorer` module identifies and refactors string concatenations in loops in Python code to improve the performance and energy efficiency of the software. It specifically handles these concatenations by, instead, adding the string for each iteration to a list that is then converted to a string, ensuring proper refactoring while maintaining the original functionality.

11.2 Uses

- Uses `Smell` interface for data access
- Inherits from `BaseRefactorer`
- Uses `astroid` library for AST manipulation

11.3 Syntax

Exported Constants: None
Exported Access Programs:

Name	In	Out	Exceptions
`UseListAccumulationRefactorer`	`output_dir: Path`	`self`	None
`refactor`	`file_path: Path, pylint_smell: Smell, initial_emissions:` $\mathbb{R}$	None	`TypeError`, `IOError`

11.4 Semantics

11.4.1 State Variables

- `target_lines: list[int]`: Line numbers requiring refactoring
- `assign_var: str`: Target concatenation variable name
- `target_node: NodeNG`: AST node of concatenation target
- `last_assign_node: Assign|AugAssign`: Last variable assignment before loop
- `concat_nodes: list[Assign|AugAssign]`: Detected concatenation nodes
- `reassignments: list[Assign]`: Variable reassignments in loop scope
- `outer_loop: For|While`: Outermost loop containing concatenations

11.4.2 Environment Variables

None

11.4.3 Assumptions

- Input files contain valid Python syntax
- Smell detection provides valid line numbers and variable names

11.4.4 Access Routine Semantics

`UseListAccumulationRefactorer(output_dir: Path)`

- **Transition**: Initializes refactorer with output directory
- **Output**: `self`
- **Exception**: None

`refactor(file_path: Path, pylint_smell: Smell, initial_emissions: R R)`

- **Transition**:
  - Parses code using `_visit` pattern for AST traversal
  - Identifies concatenations with `_find_reassignments`
  - Determines scope via `_find_scope` and `_find_last_assignment`
  - Generates temp names with `_generate_temp_list_name`
  - Modifies code using `_add_node_to_body`

  - Validates transformations before writing to refactored file

- **Output**: None
- **Exception**:
  - `IOError`: File read/write failures
  - `TypeError`: AST parsing errors

11.4.5 Local Functions

`_visit(node: NodeNG)`

- **Transition**: Collects concatenation nodes and loop structures
- **Output**: None
- **Exception**: None

`_find_reassignments()`

- **Transition**: Finds variable reassignments in loop scope
- **Output**: None
- **Exception**: None

`_find_last_assignment(scope: NodeNG)`

- **Transition**: Locates final variable assignment before loop
- **Output**: None
- **Exception**: Raises `TypeError` for invalid scope

`_find_scope()`

- **Transition**: Determines insertion point for list initialization
- **Output**: None
- **Exception**: Requires `concat_nodes` to be populated

`_generate_temp_list_name()`

- **Transition**: Creates unique list name for complex targets
- **Output**: Returns generated name string
- **Exception**: Raises `TypeError` for unsupported nodes

`_add_node_to_body(code_file: str, nodes_to_change: list[tuple])`

- **Transition**:
  - Replaces concatenations with list operations
  - Adds join() call and list initialization
- **Output**: Returns modified source code
- **Exception**: Requires valid node references

12 MIS of MakeMethodStaticRefactorer

`MakeStaticRefactorer`

12.1 Module

The `MakeStaticRefactorer` module identifies and refactors class methods that don't make use of their instance attributes to improve the readability, performance and energy efficiency of the software. It specifically handles these methods by turning them into static functions and ensuring any calls to this method use the proper calling syntax. This ensures proper refactoring while maintaining the original functionality.

12.2 Uses

`BaseRefactorer`

12.3 Syntax

Exported Constants: None

Exported Access Programs:

Name	In	Out	Exceptions
`apply_transformation`	`node: AST`	`AST`	None
`identify_methods`	`node: AST`	`List[Method]`	None
`analyze_method_usage`	`method: Method`	`bool`	None
`transform_to_static`	`method: Method`	`Method`	None

12.4 Semantics

12.4.1 State Variables

- `class_hierarchy`: Dictionary mapping classes to their inheritance tree

- `method_registry`: Dictionary mapping methods to their usage patterns
- `static_candidates`: Set of methods eligible for static conversion

12.4.2 Environment Variables

Inherits from `BaseRefactorer`

12.4.3 Assumptions

- Class inheritance hierarchies are well-defined
- Method definitions are complete and valid
- No dynamic method creation or modification

12.4.4 Access Routine Semantics

`apply_transformation(node: AST)`

- **Transition**:
  - Identifies candidate methods using `identify_methods`
  - Analyzes each method using `analyze_method_usage`
  - Transforms eligible methods using `transform_to_static`
  - Updates method calls using `_update_method_calls`
  - Validates transformation using `_validate_transformation`
  - Reconstructs code with static methods
- **Output**: Modified AST
- **Exception**: None

`identify_methods(node: AST)`

- **Transition**: Analyzes AST for instance methods using `_build_class_hierarchy`
- **Output**: List of identified methods
- **Exception**: None

`analyze_method_usage(method: Method)`

- **Transition**: Evaluates method for static conversion using `_check_inheritance_safety`
- **Output**: Boolean indicating if method can be made static
- **Exception**: None

`transform_to_static(method: Method)`

- **Transition**: Converts instance method to static using `_update_method_calls` and validates using `_validate_transformation`
- **Output**: Transformed method
- **Exception**: None

12.4.5 Local Functions

`_build_class_hierarchy(node: AST)`

- **Transition**: Analyzes class definitions and builds inheritance relationships
- **Output**: Dictionary mapping classes to their parent classes
- **Exception**: None

`_check_inheritance_safety(method: Method)`

- **Transition**: Verifies method can be safely made static across inheritance chain
- **Output**: Boolean indicating if transformation is safe
- **Exception**: None

`_update_method_calls(old_method: str, new_method: str)`

- **Transition**: Updates all call sites to use static method syntax
- **Output**: None
- **Exception**: None

`_validate_transformation(method: Method)`

- **Transition**: Verifies the static method transformation maintains functionality
- **Output**: Boolean indicating if transformation is valid
- **Exception**: None

13 MIS of LongElementChainRefactorer

`LongElementChainRefactorer`

13.1 Module

LongElementChainRefactorer is a module that refactors long element chains, specifically focusing on flattening nested dictionaries to improve readability, maintainability, and energy efficiency. The module uses a recursive flattening strategy while caching previously seen patterns for optimization.

13.2 Uses

`BaseRefactorer`

13.3 Syntax

13.3.1 Exported Constants

None

13.3.2 Exported Access Programs

Name	In	Out	Exceptions
`apply_transformation`	`node: AST`	`AST`	None
`identify_nested_dicts`	`node: AST`	`List[Dict]`	None
`flatten_dict`	`dict_node: Dict`	`Dict`	None
`update_access_calls`	`old_path: str, new_path: str`	None	None

13.4 Semantics

13.4.1 State Variables

- **dict_patterns**: Dictionary mapping access patterns to their flattened forms
- **flattened_keys**: Set of generated flattened key names
- **reference_map**: Mapping of original to flattened dictionary paths

13.4.2 Environment Variables

Inherits from `BaseRefactorer`

13.4.3 Assumptions

- Dictionary keys are valid Python identifiers
- No key name conflicts in flattened structure
- Dictionary access patterns are consistent

13.4.4 Access Routine Semantics

`apply_transformation(node: AST)`

- **Transition**:
  - Identifies nested dictionaries using `identify_nested_dicts`
  - Flattens each dictionary using `flatten_dict`
  - Updates access patterns using `update_access_calls`
  - Updates code references using `_update_references`
  - Validates dictionary access using `_validate_dict_access`
  - Reconstructs code with flattened structure
- **Output**: Modified AST
- **Exception**: None

`identify_nested_dicts(node: AST)`

- **Transition**: Analyzes AST for nested dictionary patterns using `_analyze_dict_complexity`
- **Output**: List of identified dictionaries
- **Exception**: None

`flatten_dict(dict_node: Dict)`

- **Transition**: Creates flattened dictionary structure using `_generate_flat_key`
- **Output**: Flattened dictionary
- **Exception**: None

`update_access_calls(old_path: str, new_path: str)`

- **Transition**: Updates dictionary access patterns using `_validate_dict_access` and `_update_references`
- **Output**: None
- **Exception**: None

13.4.5 Local Functions

`_analyze_dict_complexity(node: Dict)`

- **Transition**: Analyzes the nesting depth and structure of dictionary nodes
- **Output**: Integer representing complexity score
- **Exception**: None

`_generate_flat_key(path: List[str])`

- **Transition**: Concatenates path components into a flattened key name
- **Output**: String representing the flattened key
- **Exception**: None

`_validate_dict_access(path: str)`

- **Transition**: Checks if dictionary access path is valid and follows conventions
- **Output**: Boolean indicating validity
- **Exception**: None

`_update_references(old_ref: str, new_ref: str)`

- **Transition**: Updates all references to the old dictionary path with the new flattened path
- **Output**: None
- **Exception**: None

14 MIS of Measurements Module

`Measurements`

14.1 Module

The MeasurementsModule is a module designed to measure and track the carbon emissions generated by executing scripts. By leveraging the CodeCarbon library, it allows developers to assess the environmental impact of their code execution. The module runs a specified Python file, tracks the associated carbon emissions during the execution, and logs the results for further analysis. It provides functionality for measuring, logging, and extracting emissions data in a structured manner to help improve energy efficiency in software development.

14.2 Uses

- Uses `CodeCarbon` library for track energy consumption
- Uses `TemporaryDirectory` to store temporary files
- Inherits from `BaseEnergyMeter`

14.3 Syntax

14.3.1 Exported Constants

None

14.3.2 Exported Access Programs

Name	In	Out	Exceptions
`Measurements`	`output_dir: Path`	`self`	None
`measure_energy`	`None`	None	CalledProcessError and FileReading exceptions

14.4 Semantics

14.4.1 State Variables

- **emissions_data**: Stores the emissions data extracted from the CSV file generated by CodeCarbon. It is populated after the energy measurement process completes successfully. The value is either a dictionary containing the last row of emissions data or `None` if no data was extracted due to an error.
- **emissions**: Raw emissions object from CodeCarbon tracker

14.4.2 Environment Variables

None

14.4.3 Assumptions

- The file at `file_path` is a valid Python script.
- The CodeCarbon tool is properly installed and configured.
- The `EmissionsTracker` can successfully execute the Python script specified by `file_path`.
- The emissions data is captured in a CSV format and can be extracted correctly.
- The temporary directories are correctly set up and accessible during execution.

14.4.4 Access Routine Semantics

`Measurements(output_dir: Path)`

- **Transition**: Initializes the energy meter with empty emissions data
- **Output**: `self`
- **Exception**: None

`measure_energy()`

- **Transition**:
  - Logs the start of the energy measurement process
  - Creates isolated temporary directory using `TemporaryDirectory`
  - Configures system temp directories through environment variables
  - Initializes CodeCarbon `EmissionsTracker` in process mode
  - Runs the script specified by file path and captures the output
  - Stops tracker and captures raw emissions data
  - Validates emissions CSV creation
  - Parses results using `extract_emissions_csv`
- **Output**:
  - Updates `emissions` with tracker results
  - Populates `emissions_data` with parsed metrics
- **Exception**:
  - `CalledProcessError`: If script execution fails
  - `FileNotFoundError`: If emissions CSV is missing

14.4.5 Local Functions

`extract_emissions_csv(csv_file_path: Path)`

- **Transition**:
  - Attempts to read CSV file using pandas
  - Extracts last measurement record
  - Converts DataFrame row to dictionary
- **Output**: Returns dictionary of metrics or `None` on error
- **Exception**: Logs pandas read errors but does not propagate them

15 MIS of PylintAnalyzer

`PylintAnalyzer`

15.1 Module

The `PylintAnalyzer` module performs static code analysis on Python files using Pylint, with additional custom checks for detecting specific code smells. It outputs detected smells in a structured format for further processing.

15.2 Uses

- Uses Python's `pylint` library for code analysis
- Uses `ast` module for parsing and analyzing abstract syntax trees
- Uses `astor` library for converting AST nodes back to source code
- Integrates with custom checkers, including `StringConcatInLoopChecker`
- Accesses configuration settings from `analyzers_config`

15.3 Syntax

Exported Constants: None
Exported Access Programs:

Name	**In**	**Out**	**Exceptions**
`PylintAnalyzer`	`file_path: Path, source_code: Module`	`self`	None
`build_pylint_options`	None	`list[str]`	None
`analyze`	None	None	`JSONDecodeError`, `Exception`
`configure_smells`	None	None	None
`filter_for_one_code_smell`	`pylint_results: list[Smell], code: str`	`list[Smell]`	None

15.4 Semantics

15.4.1 State Variables

- `file_path: Path`: The path to the Python file being analyzed
- `source_code: Module`: The parsed abstract syntax tree of the source file
- `smells_data: list[dict]`: List of detected code smells in dictionary format

15.4.2 Environment Variables

None

15.4.3 Assumptions

- The input file is valid Python code and can be parsed into an AST
- Configuration settings (extra Pylint options, custom smell definitions) are valid

15.4.4 Access Routine Semantics

`PylintAnalyzer(file_path: Path, source_code: Module)`

- **Transition**: Initializes analyzer with file path and AST
- **Output**: `self`
- **Exception**: None

`build_pylint_options()`

- **Transition**: Constructs Pylint options list from config
- **Output**: List of option strings
- **Exception**: None

`analyze()`

- **Transition**:
  - Executes Pylint analysis with custom checks using the local detect functions
  - Populates `smells_data` with results and uses `parse_line`
- **Output**: None
- **Exception**:
  - `JSONDecodeError` for invalid Pylint output
  - `Exception` for general runtime errors

`configure_smells()`

- **Transition**: Filters `smells_data` to configured smells
- **Output**: None
- **Exception**: None

`filter_for_one_code_smell(pylint_results: list[Smell], code: str)`

- **Transition**: Filters results by smell type code
- **Output**: Filtered list of smells
- **Exception**: None

15.4.5 Local Functions

`_detect_long_message_chain(threshold?: int)`

- **Transition**: Identifies method chains exceeding length threshold
- **Output**: List of long chain smells
- **Exception**: None

`_detect_long_lambda_expression(threshold_length?: int, threshold_count?: int)`

- **Transition**: Detects oversized lambda expressions
- **Output**: List of lambda smells
- **Exception**: None

`_detect_long_element_chain(threshold?: int)`

- **Transition**: Finds long dictionary access chains
- **Output**: List of element chain smells
- **Exception**: None

`_detect_repeated_calls(threshold?: int)`

- **Transition**: Identifies excessive repeated calls
- **Output**: List of repetition smells
- **Exception**: None

`_parse_line(file_path: Path, line: int)`

- **Transition**: Extracts AST node from specific line
- **Output**: Parsed AST node
- **Exception**: None

_get_lambda_code(lambda_node: Lambda)

- **Transition**: Converts lambda node to source code
- **Output**: String representation of lambda
- **Exception**: None

16 MIS of UseAGeneratorRefactorer

UseAGeneratorRefactorer

16.1 Module

The `UseAGeneratorRefactorer` module identifies and refactors unnecessary list comprehensions in Python code by converting them to generator expressions. This refactoring improves energy efficiency while maintaining the original functionality.

16.2 Uses

- Uses `Smell` interface for data access
- Inherits from `BaseRefactorer`
- Uses Python's `ast` module for parsing and manipulating abstract syntax trees

16.3 Syntax

Exported Constants: None
Exported Access Programs:

Name	In	Out	Exceptions
`UseAGeneratorRefactorer`	`output_dir: Path`	`self`	None
`refactor`	`file_path: Path,` `pylint_smell: Smell,` `initial_emissions:` $\mathbb{R}$	None	`IOError`, `TypeError`

16.4 Semantics

16.4.1 State Variables

- `temp_dir: Path`: Directory path for storing refactored files
- `output_dir: Path`: Directory path for saving final refactored code

16.4.2 Environment Variables

None

16.4.3 Assumptions

- The input file contains valid Python syntax
- `pylint_smell` provides valid line/column locations

16.4.4 Access Routine Semantics

`UseAGeneratorRefactorer(output_dir: Path)`

- **Transition**: Initializes temporary directory within output directory
- **Output**: `self`
- **Exception**: None

`refactor(file_path: Path, pylint_smell: Smell, initial_emissions: ℝ)`

- **Transition**:
  - Reads source code using `ListCompInAnyAllTransformer` metadata
  - Applies AST transformation via `_replace_node` for node substitution
  - Uses `leave_Call` in transformer to identify replacement targets
  - Validates and writes modified code using generator expressions
- **Output**: None
- **Exception**:
  - `IOError`: File read/write failures
  - `TypeError`: CST parsing errors

16.4.5 Local Functions

`_replace_node(tree: Module, old_node: ListComp, new_node: GeneratorExp)`

- **Transition**: Replaces list comprehension node with generator expression
- **Output**: Modified AST
- **Exception**: None

`ListCompInAnyAllTransformer`

- **Transition**: Custom CST transformer that identifies and converts list comprehensions in any()/all() calls
- **Output**: Modified syntax tree
- **Exception**: None

17 MIS of CacheRepeatedCallsRefactorer

`CacheRepeatedCallsRefactorer`

17.1 Module

The `CacheRepeatedCallsRefactorer` identifies and caches repeated function calls using temporary variables to improve performance and energy efficiency while preserving functionality.

17.2 Uses

- Uses `Smell` interface for data access
- Inherits from `BaseRefactorer`
- Uses Python's `ast` module for AST manipulation

17.3 Syntax

Exported Constants: None
Exported Access Programs:

Name	**In**	**Out**	**Exceptions**
`CacheRepeatedCallsRefactorer`	`output_dir: Path`	`self`	None
`refactor`	`file_path: Path, pylint_smell: Smell, initial_emissions:` $\mathbb{R}$	None	`IOError`, `TypeError`

17.4 Semantics

17.4.1 State Variables

- `cached_var_name: str`: Name of the temporary variable used for caching.
- `target_line: int`: Line number where refactoring is applied.

17.4.2 Environment Variables

None

17.4.3 Assumptions

- Input files contain valid Python syntax
- Smell detection provides valid call patterns
- Repeated calls have no side effects

17.4.4 Access Routine Semantics

`CacheRepeatedCallsRefactorer(output_dir: Path)`

- **Transition**: Initializes temp directory
- **Output**: `self`
- **Exception**: None

`refactor(file_path: Path, smell: CRCSmell, ...)`

- **Transition**:
  - Generates cache name via `_extract_function_name`
  - Locates insertion point with `_find_insert_line`
  - Determines indentation via `_get_indentation`
  - Modifies calls using `_replace_call_in_line`
  - Validates scope with `_find_valid_parent`
- **Output**: None
- **Exception**:
  - `IOError`: File access failures
  - `TypeError`: Invalid AST structure

17.4.5 Local Functions

`_extract_function_name(call_string: str)`

- **Transition**: Extracts function name from call expression
- **Output**: String containing function name
- **Exception**: None

`_get_indentation(lines: list[str], line_number: int)`

- **Transition**: Calculates whitespace for code insertion
- **Output**: Indentation string
- **Exception**: None

`_replace_call_in_line(line: str, call_string: str, cached_var_name: str)`

- **Transition**: Replaces function calls with cache variable
- **Output**: Modified source line
- **Exception**: None

`_find_valid_parent(tree: ast.Module)`

- **Transition**: Locates common parent for all call instances
- **Output**: Parent AST node or None
- **Exception**: None

`_find_insert_line(parent_node: ast.AST)`

- **Transition**: Determines optimal insertion point
- **Output**: Line number for cache assignment
- **Exception**: None

`_line_in_node_body(node: ast.AST, line: int)`

- **Transition**: Verifies line belongs to node body
- **Output**: Boolean existence check
- **Exception**: None

18 MIS of PluginInitiator

18.1 Module

`PluginInitiator` is a module that initialises the VS Code plugin and registers commands for VS Code Plugin.

18.2 Uses

- `SmellDetector` to register and manage smell detection functionality.
- `RefactorManager` to register and manage refactoring operations.
- `FilterManager` to register and manage smell filtering operations.

18.3 Syntax

Exported Constants:

- `ecoOutput`: VS Code OutputChannel for logging and information display

Exported Access Programs:

Name	In	Out	Exceptions
`activate`	`context: vscode.ExtensionContext`	void	None
`deactivate`	None	void	None
`isSmellLintingEnabled`	None	boolean	None

18.4 Semantics

18.4.1 State Variables

- `smellLintingEnabled: boolean` - Tracks if smell linting is enabled
- `backendLogManager: LogManager` - Manages backend logging

18.4.2 Environment Variables

- VS Code extension context
- Workspace configuration state

18.4.3 Assumptions

- The plugin is correctly loaded in VS Code
- VS Code APIs are available and accessible
- Required modules (SmellDetector, RefactorManager, FilterManager) are properly initialised

18.4.4 Access Routine Semantics

`activate(context: vscode.ExtensionContext)`

- **Transition**:
  - Initialises output channel for logging
  - Initialises SmellDetector module
  - Initialises RefactorManager module
  - Initialises FilterManager module
  - Sets up workspace configuration
  - Registers plugin commands
- **Output**: None
- **Exception**: None

`deactivate()`

- **Transition**: Cleans up resources and stops background processes
- **Output**: None
- **Exception**: None

`isSmellLintingEnabled()`

- **Transition**: Returns current state of smell linting
- **Output**: boolean indicating if smell linting is enabled
- **Exception**: None

18.4.5 Local Functions

None

19 MIS of BackendCommunicator

19.1 Module

`BackendCommunicator` handles all communication between the plugin and the backend service. It sends requests for analysis or refactoring and receives results.

19.2 Uses

- `EcoOptimizer` for executing backend service operations and health checks

19.3 Syntax

Exported Constants:

- BASE_URL: string - Base URL for backend API endpoints

Exported Access Programs:

Name	In	Out	Exceptions
checkServerStatus	None	Promise<void>	Network Error
initLogs	log_dir: string	Promise<boolean>	Network Error
fetchSmells	filePath: string, enabledSmells: Record<string, Record<string, number | string>>	Promise<{smells: Smell[], status: number}>	Network Error
backendRefactor Smell	smell: Smell, workspacePath: string	Promise<RefactoredData>	Network Error
backendRefactor SmellType	smell: Smell, workspacePath: string	Promise<RefactoredData>	Network Error

19.4 Semantics

19.4.1 State Variables

- serverStatus: ServerStatusType - Tracks the backend server's current status

19.4.2 Environment Variables

- SERVER_URL: string - Backend server URL from environment configuration

19.4.3 Assumptions

- EcoOptimizer backend service is reachable and operational
- Network connection is available for API communication
- Valid workspace configuration exists for operations requiring paths
- Input files contain valid Python syntax

19.4.4 Access Routine Semantics

`checkServerStatus()`

- **Transition**: Verifies backend service availability and updates extension status
- **Output**: Promise resolving to void
- **Exception**: Logs network errors and updates server status to DOWN

`initLogs(log_dir: string)`

- **Transition**: Initialises and synchronises logs with the backend server
- **Output**: Promise<boolean> indicating success or failure
- **Exception**: Logs initialisation errors and returns false

`fetchSmells(filePath: string, enabledSmells: Record)`

- **Transition**: Analyses source code for code smells using backend detection service
- **Output**: Promise resolving to smell detection results and HTTP status
- **Exception**: Throws Error for network failures or invalid responses

`backendRefactorSmell(smell: Smell, workspacePath: string)`

- **Transition**: Executes code refactoring for a specific detected smell pattern
- **Output**: Promise resolving to refactoring result data
- **Exception**: Throws Error for invalid workspace path or refactoring failures

`backendRefactorSmellType(smell: Smell, workspacePath: string)`

- **Transition**: Refactors all smells of a specific type in a file
- **Output**: Promise resolving to refactoring result data
- **Exception**: Throws Error for invalid workspace path or refactoring failures

19.4.5 Local Functions

None

20 MIS of SmellDetector

20.1 Module

`SmellDetector` analyses Python files for code smells and manages the detection workflow.

20.2 Uses

- `BackendCommunicator` for communicating with SourceCodeOptimizer
- `CacheManager` for managing smell detection results
- `FileHighlighter` for highlighting detected smells
- `EventManager` for managing detection events
- `HoverManager` for displaying smell information

20.3 Syntax

Exported Constants: None

Exported Access Programs:

Name	In	Out	Exceptions
`detectSmellsFile`	`filePath: string, smellsViewProvider: SmellsViewProvider, cacheManager: CacheManager`	Promise<void>	Analysis Error
`detectSmellsFolder`	`folderPath: string, smellsViewProvider: SmellsViewProvider, cacheManager: CacheManager`	Promise<void>	Analysis Error

20.4 Semantics

20.4.1 State Variables

- `serverStatus`: ServerStatusType - Current status of the backend server

20.4.2 Environment Variables

- `enabledSmells`: Record<string, SmellConfig> - Configuration of enabled smell detectors

20.4.3 Assumptions

- Target files have valid Python syntax
- Backend server is operational for non-cached analysis
- At least one smell detector is enabled in settings
- Valid workspace configuration exists

20.4.4 Access Routine Semantics

`detectSmellsFile(filePath, smellsViewProvider, cacheManager)`

- **Transition**: Checks a Python file using `precheckAndMarkQueued()`, analyses it for code smells, updates cache and view
- **Output**: Promise resolving to void
- **Exception**: Throws error if analysis fails

`detectSmellsFolder(folderPath, smellsViewProvider, cacheManager)`

- **Transition**: Recursively analyzes Python files in directory, shows progress
- **Output**: Promise resolving to void
- **Exception**: Throws error if folder scanning or analysis fails

20.4.5 Local Functions

`precheckAndMarkQueued(filePath, smellsViewProvider, cacheManager)`

- **Transition**: Validates conditions before analysis and manages file status
- **Output**: Promise<boolean> indicating if analysis should proceed
- **Exception**: None

21 MIS of SmellRefactorer

21.1 Module

`SmellRefactorer` applies a refactoring to a detected smell.

21.2 Uses

- `BackendCommunicator` for sending the smell data to Source Code Optimizer for refactoring.
- `RefactorManager` for managing refactoring workflows.

21.3 Syntax

Exported Constants: None

Exported Access Programs:

Name	In	Out	Exception
`refactor`	smell: Smell	None	Invalid input

21.4 Semantics

21.4.1 State Variables

None

21.4.2 Environment Variables

None

21.4.3 Assumptions

- The smell data is valid and correctly identifies a refactorable issue.

21.4.4 Access Routine Semantics

`refactor(smell: Smell)`

- **Transition**: Sends the smell data to the backend for refactoring and applies the changes in the editor.
- **Output**: None.
- **Exception**: Logs errors for invalid inputs or failed refactoring.

21.4.5 Local Functions

None

22 MIS of FileHighlighter

22.1 Module

`FileHighlighter` is a module that manages highlighting of code regions in the VS Code editor.

22.2 Uses

- `ViewProvider` for managing editor decorations and visual updates

22.3 Syntax

Exported Constants: None

Exported Access Programs:

Name	In	Out	Exception
`getInstance`	`cacheManager: CacheManager`	FileHighlighter	None
`updateHighlightsForVisibleEditors`	None	void	None
`resetHighlights`	None	void	None
`highlightSmells`	`editor: vscode.TextEditor`	void	None

22.4 Semantics

22.4.1 State Variables

- `instance`: FileHighlighter - Singleton instance of the highlighter
- `decorations`: TextEditorDecorationType[] - Active editor decorations
- `cacheManager`: CacheManager - Reference to the cache manager

22.4.2 Environment Variables

- `smellsColours`: Configuration for smell highlight colours
- `useSingleColour`: Boolean flag for using single highlight colour
- `singleHighlightColour`: Colour value for single highlight mode
- `highlightStyle`: Style configuration for highlights

22.4.3 Assumptions

- `CacheManager` is properly initialized
- Valid configuration exists for highlight colours and styles

22.4.4 Access Routine Semantics

`getInstance(cacheManager: CacheManager)`

- **Transition**: Creates or returns singleton instance of `FileHighlighter`
- **Output**: `FileHighlighter` instance
- **Exception**: None

`updateHighlightsForVisibleEditors()`

- **Transition**: Updates highlights for all visible Python files using `_getDecoration` and `_updateHighlightsForFile`
- **Output**: None
- **Exception**: None

`resetHighlights()`

- **Transition**: Removes all active highlights from editors
- **Output**: None
- **Exception**: None

`highlightSmells(editor: vscode.TextEditor)`

- **Transition**: Applies highlights for cached smells in the editor
- **Output**: None
- **Exception**: None

22.4.5 Local Functions

`_getDecoration(colour, style)`

- **Transition**: Creates decoration type based on configuration
- **Output**: TextEditorDecorationType object
- **Exception**: None

`_updateHighlightsForFile(filePath)`

- **Transition**: Updates highlights for a specific file
- **Output**: None
- **Exception**: None

23 MIS of HoverManager

23.1 Module

`HoverManager` manages hover effects to display contextual information.

23.2 Uses

- `ViewProvider` for managing hover display and updates

23.3 Syntax

Exported Constants: None

Exported Access Programs:

Name	In	Out	Exception
`register`	`context: vscode.ExtensionContext`	void	None
`provideHover`	`document: TextDocument, position: Position, token: CancellationToken`	ProviderResult<Hover>	None

23.4 Semantics

23.4.1 State Variables

- `cacheManager`: CacheManager - Reference to the cache manager

23.4.2 Environment Variables

None

23.4.3 Assumptions

- VS Code editor is active with a Python file open
- Valid hover position is provided

23.4.4 Access Routine Semantics

`register(context: vscode.ExtensionContext)`

- **Transition**: Registers hover provider for Python files
- **Output**: None
- **Exception**: None

`provideHover(document, position, token)`

- **Transition**: Provides hover content with smell information
- **Output**: Hover content or undefined
- **Exception**: None

23.4.5 Local Functions

None

24 MIS of RefactorManager

24.1 Module

`RefactorManager` manages the process of applying refactorings to detected smells.

24.2 Uses

- `EnergyMetrics` for tracking refactoring impact
- `CacheManager` for managing refactoring state
- `EventManager` for handling refactoring events

24.3 Syntax

Exported Constants: None

Exported Access Programs:

Name	In	Out	Exception
`register`	`context: vscode.ExtensionContext`	void	None
`provideHover`	`document: TextDocument, position: Position, token: CancellationToken`	ProviderResult<Hover>	None

24.4 Semantics

24.4.1 State Variables

- `serverStatus`: ServerStatusType - Current status of the backend server

24.4.2 Environment Variables

- `WORKSPACE_CONFIGURED_PATH`: string - Path to workspace configuration

24.4.3 Assumptions

- Backend server is operational
- Valid workspace configuration exists

- Smell data is valid and complete
- Required view providers are initialized

24.4.4 Access Routine Semantics

`refactor(smellsViewProvider, refactoringDetailsViewProvider, smell, context, isRefactorA`

- **Transition**: Orchestrates complete refactoring workflow
- **Output**: Promise resolving to void
- **Exception**: Throws error for validation failures

`startRefactorSession(smell, refactoredData, refactoringDetailsViewProvider)`

- **Transition**: Initialises and manages refactoring session
- **Output**: Promise resolving to void
- **Exception**: None

24.4.5 Local Functions

None

25 MIS of CacheManager

25.1 Module

`CacheManager` manages caching of detected smells and file states.

25.2 Uses

- `BackendCommunicator` for backend communication
- `ViewProvider` for UI updates

25.3 Syntax

Exported Constants: None

Exported Access Programs:

Name	**In**	**Out**	**Exception**
`setCachedSmells`	`filePath: string`, `smells: Smell[]`	Promise<void>	None
`getCachedSmells`	`filePath: string`	Smell[] or undefined	None
`hasCachedSmells`	`filePath: string`	boolean	None
`clearAllCachedSmells`	None	Promise<void>	None

25.4 Semantics

25.4.1 State Variables

- `cacheUpdatedEmitter`: EventEmitter - Emits cache update events
- `fileStatuses`: Map<string, string> - Tracks file statuses
- `fileSmells`: Map<string, Smell[]> - Stores cached smells

25.4.2 Environment Variables

None

25.4.3 Assumptions

- VS Code extension context is valid
- File paths are normalized
- Smell data structure is consistent

25.4.4 Access Routine Semantics

`setCachedSmells(filePath, smells)`

- **Transition**: Stores smells in cache for specified file using `generateSmellId()`
- **Output**: Promise resolving to void
- **Exception**: None

`getCachedSmells(filePath)`

- **Transition**: Retrieves cached smells for file using `hasCachedSmells()`
- **Output**: Array of smells or undefined
- **Exception**: None

`hasCachedSmells(filePath)`

- **Transition**: Checks if file has cached smells
- **Output**: boolean
- **Exception**: None

`clearAllCachedSmells()`

- **Transition**: Removes all cached smells
- **Output**: Promise resolving to void
- **Exception**: None

25.4.5 Local Functions

- `generateSmellId(smell)`: Generates unique ID for smell
- `generateFileHash(filePath)`: Generates hash for file content

26 MIS of FilterManager

26.1 Module

`FilterManager` manages filtering of detected smells.

26.2 Uses

- `CacheManager` for accessing smell data
- `EventManager` for handling filter events
- `HoverManager` for displaying filtered smells

26.3 Syntax

Exported Constants: None

Exported Access Programs:

Name	**In**	**Out**	**Exception**
`toggleSmellFilter`	`smellKey: string`	void	None
`editSmellFilterOption`	`item: {smellKey: string, optionKey: string, value: any}`	Promise<void>	Validation Error
`refresh`	None	void	None

26.4 Semantics

26.4.1 State Variables

- `filterState`: Map<string, boolean> - Tracks filter states
- `filterOptions`: Map<string, any> - Stores filter options

26.4.2 Environment Variables

None

26.4.3 Assumptions

- Valid smell keys are provided
- Filter options are properly configured
- Cache manager is initialized

26.4.4 Access Routine Semantics

`toggleSmellFilter(smellKey)`

- **Transition**: Toggles filter state for smell
- **Output**: None
- **Exception**: None

`editSmellFilterOption(item)`

- **Transition**: Updates filter option value
- **Output**: Promise resolving to void
- **Exception**: Throws validation error for invalid input

`refresh()`

- **Transition**: Updates filter display
- **Output**: None
- **Exception**: None

26.4.5 Local Functions

None

27 MIS of EnergyMetrics

27.1 Module

`EnergyMetrics` manages energy consumption measurements and metrics display.

27.2 Uses

- `ViewProvider` for displaying metrics

27.3 Syntax

Exported Constants: None

Exported Access Programs:

Name	In	Out	Exception
`updateMetrics`	`filePath: string, carbonSaved: number, smellSymbol: string`	void	None
`refresh`	None	void	None
`getTreeItem`	`element: MetricTreeItem`	TreeItem	None

27.4 Semantics

27.4.1 State Variables

- `metricsData`: Record<string, MetricsDataItem> - Stores metrics data
- `onDidChangeTreeData`: EventEmitter - Emits tree data change events

27.4.2 Environment Variables

None

27.4.3 Assumptions

- Valid file paths are provided
- Metrics data is properly formatted
- Tree view is initialized

27.4.4 Access Routine Semantics

`updateMetrics(filePath, carbonSaved, smellSymbol)`

- **Transition**: Updates metrics for file and smell using `calculateFileMetrics()`
- **Output**: None
- **Exception**: None

`refresh()`

- **Transition**: Updates metrics display using `formatNumber()`
- **Output**: None
- **Exception**: None

`getTreeItem(element)`

- **Transition**: Creates tree item for metrics display
- **Output**: TreeItem
- **Exception**: None

27.4.5 Local Functions

- `calculateFileMetrics(filePath, metricsData)`: Calculates metrics for file
- `formatNumber(number)`: Formats number for display

28 MIS of ViewProvider

28.1 Module

`ViewProvider` manages the display of smells and metrics in VS Code.

28.2 Uses

None

28.3 Syntax

Exported Constants: None

Exported Access Programs:

Name	**In**	**Out**	**Exception**
`refresh`	None	void	None
`setStatus`	`filePath: string, status: string`	void	None
`setSmells`	`filePath: string, smells: Smell[]`	void	None
`removeFile`	`filePath: string`	boolean	None

28.4 Semantics

28.4.1 State Variables

- `fileStatuses`: Map<string, string> - Tracks file statuses
- `fileSmells`: Map<string, Smell[]> - Stores smell data
- `onDidChangeTreeData`: EventEmitter - Emits tree data change events

28.4.2 Environment Variables

None

28.4.3 Assumptions

- VS Code extension context is valid
- File paths are normalized
- Smell data structure is consistent

28.4.4 Access Routine Semantics

`refresh()`

- **Transition**: Updates tree view display
- **Output**: None
- **Exception**: None

`setStatus(filePath, status)`

- **Transition**: Updates file status in view
- **Output**: None
- **Exception**: None

`setSmells(filePath, smells)`

- **Transition**: Updates smell data in view
- **Output**: None
- **Exception**: None

`removeFile(filePath)`

- **Transition**: Removes file from view
- **Output**: boolean indicating success
- **Exception**: None

28.4.5 Local Functions

None

29 MIS of EventManager

29.1 Module

`EventManager` manages event handling and propagation.

29.2 Uses

- `ViewProvider` for UI updates

29.3 Syntax

Exported Constants: None

Exported Access Programs:

Name	In	Out	Exception
`getStatus`	None	ServerStatusType	None
`setStatus`	`newStatus: ServerStatusType`	void	None
`on`	`event: string, listener: Function`	void	None
`emit`	`event: string, data: any`	void	None

29.4 Semantics

29.4.1 State Variables

- `status`: ServerStatusType - Current server status
- `eventEmitter`: EventEmitter - Event emitter instance

29.4.2 Environment Variables

None

29.4.3 Assumptions

- Event listeners are properly registered
- Event types are consistent
- Server status is valid

29.4.4 Access Routine Semantics

`getStatus()`

- **Transition**: Returns current server status
- **Output**: ServerStatusType
- **Exception**: None

`setStatus(newStatus)`

- **Transition**: Updates server status and emits event

- **Output**: None
- **Exception**: None

`on(event, listener)`

- **Transition**: Registers event listener
- **Output**: None
- **Exception**: None

`emit(event, data)`

- **Transition**: Emits event with data
- **Output**: None
- **Exception**: None

29.4.5 Local Functions

None

30 Appendix — Reflection

The information in this section will be used to evaluate the team members on the graduate attribute of Problem Analysis and Design.

The purpose of reflection questions is to give you a chance to assess your own learning and that of your group as a whole, and to find ways to improve in the future. Reflection is an important part of the learning process. Reflection is also an essential component of a successful software development process.

Reflections are most interesting and useful when they're honest, even if the stories they tell are imperfect. You will be marked based on your depth of thought and analysis, and not based on the content of the reflections themselves. Thus, for full marks we encourage you to answer openly and honestly and to avoid simply writing "what you think the evaluator wants to hear."

Please answer the following questions. Some questions can be answered on the team level, but where appropriate, each team member should write their own response:

Group Reflection

1. *Which of your design decisions stemmed from speaking to your client(s) or a proxy (e.g. your peers, stakeholders, potential users)? For those that were not, why, and where did they come from?*

   The decision to modularize the refactorers into specific "smell-focused" components was largely inspired by a conversation with our supervisor, who is also our primary stakeholder. During one of our discussions, our supervisor suggested that the problem at hand had the potential to evolve into a graduate-level reinforcement learning project. This idea of managing multiple refactoring strategies and selecting the best one based on certain conditions led to the insight that organizing the refactorers by the specific types of code smells they address would make the system more extensible. By focusing each component on a particular code smell, we could later build upon the design and possibly incorporate machine learning or reinforcement learning strategies to optimize refactorer selection. This modular approach would allow for easier integration of additional strategies in the future, making the tool scalable as the project evolves.

   Another important design decision influenced by our supervisor was the idea to validate the refactored code using a test suite. Our supervisor emphasized that in a real-world application, validating the integrity of the refactored code with a comprehensive test suite was a crucial step.

   Both of these design decisions were informed by valuable input from our supervisor, ensuring that the project stayed grounded in real-world applicability and allowed for future enhancements and improvements.

2. *While creating the design doc, what parts of your other documents (e.g. requirements, hazard analysis, etc), it any, needed to be changed, and why?*

While creating the design document, several components of the project were revised to improve clarity and focus. Specifically, the list of code smells targeted by the refactoring library was refined by adding new smells that align more closely with our sustainability goals and removing others deemed less impactful. This required updates to the requirements document to ensure it accurately reflected the new scope of supported refactorings. Additionally, the decision was made to remove the metric reporting functionality due to its complexity and limited time, which led to corresponding modifications in both the requirements document and the VnV plan, where this feature had previously been considered for validation. Moreover, the reinforcement learning model, initially intended to optimise refactoring decisions, was excluded from the project due to time constraints and implementation challenges. This necessitated updates to the hazard analysis document to remove risks associated with this component and to better align the analysis with the reduced project scope. These changes ensure consistency and maintain a realistic and achievable project timeline.

3. *What are the limitations of your solution? Put another way, given unlimited resources, what could you do to make the project better? (LO_ProbSolutions)*

   The energy measurement library we selected, Codecarbon, proved to be less reliable than anticipated, which affects the accuracy of some of our results. Ideally, we would replace it with a more dependable resource. However, due to time constraints and the inherent complexity of measuring CO_2 emissions from code, this isn't feasible within the scope of this project. For now, we are assuming Codecarbon's reliability. In a real-world implementation, we would prioritize using a more robust energy measurement system.

4. *Give a brief overview of other design solutions you considered. What are the benefits and tradeoffs of those other designs compared with the chosen design? From all the potential options, why did you select the documented design? (LO_Explores)*

   We considered incorporating a machine learning aspect into the project, specifically using reinforcement learning (RL) to manage the refactoring process. The idea was to treat the selection and application of refactoring strategies as a decision-making process, where an agent could learn the best strategies over time based on rewards and outcomes.

   In this approach, the agent would represent the system that applies different refactoring techniques to the code. The environment would be the code itself, with various code smells and inefficiencies that the agent needs to address. The actions the agent would take would involve selecting and applying one of the predefined refactoring strategies (like long lambda function or long parameter list). The reward would be the resulting decrease in energy consumption (i.e., reduction in CO_2 emissions), measured after the code is refactored and executed. The agent would receive a positive reward for actions that successfully lead to more energy-efficient code and a negative reward for actions that increase energy consumption. Over time, the agent would learn to prioritize and apply the most effective refactoring techniques based on the rewards it receives.

While this machine learning solution seemed promising, there were a few trade-offs to consider. First, implementing reinforcement learning would significantly increase the complexity of the project. It would require training data, fine-tuning the agent's learning parameters, and ensuring that the agent's actions actually lead to measurable improvements in CO2 efficiency. Additionally, RL would require ongoing iteration to improve its performance, which could be time-consuming and resource-intensive, especially given the limited time available for the project.

Another concern was that reinforcement learning, while powerful, might not always be the most effective or efficient solution for this kind of task. The selection of refactoring strategies is not necessarily a highly complex decision-making process that requires learning over time. Since we already have a set of predefined strategies, a more direct, rule-based approach was more appropriate. We could achieve the same results without the need for training the agent or dealing with the unpredictability of machine learning models.

Given these trade-offs, we opted to stick with the more straightforward approach of selecting and applying refactoring strategies based on predefined rules. This decision was driven by the need for a practical and efficient solution within the given project constraints. While reinforcement learning could be an interesting exploration for future versions of the tool, the current design provides a reliable and manageable way to achieve the desired results without adding unnecessary complexity.

Mya Hussain

1. *What went well while writing this deliverable?*

   Writing the deliverable helped to clearly decompose the system into manageable modules. This ensured no functionality was missed in the implementation process and that all components connected in a way that made sense.

2. *What pain points did you experience during this deliverable, and how did you resolve them?*

   It was strange that we had already coded the project before completing this deliverable. It acted as more of a sanity check that our design decisions made sense rather than an actual blueprint of what to do. This made this deliverable easier to write as the code was already present but also made the work feel unnecessarily redundant i.e boring to do. It often felt like I was documenting things that were already clear or implemented. This repetition made the process less engaging and, at times, a bit tedious. To resolve this, I focused on framing the document as an opportunity to validate and formalize our design decisions, which helped shift the mindset from simply checking off tasks to reaffirming the thought process behind our choices.

Sevhena Walker

1. *What went well while writing this deliverable?*

Our team already had a pretty solid idea of how we wanted to break up our system, as well as the key components that should be involved, even before we started working on the MG and MIS documents. We had already coded a decent portion of the system and, in doing so, had explored and tested various design approaches and options. This hands-on experience gave us a strong foundation and a practical understanding of what worked and what didn't, which significantly influenced our final design choices. For example, we had already determined that the refactorers would be structured as individual classes inheriting from a common base class, which simplified documenting shared functionality in the MIS.

2. *What pain points did you experience during this deliverable, and how did you resolve them?*

   One of the biggest pain points was turning our informal design ideas and code into well-defined, modular components with clear inputs, outputs, and semantics. We had to carefully review the existing code to make sure the documentation matched its behaviour while keeping things flexible for future changes. We also ran into some inconsistencies that required minor refactoring to clean up our interfaces. Another tricky part was finding the right balance between providing enough detail and keeping the documentation readable without going too deep into implementation. We tackled these problems by reviewing everything multiple times, getting feedback, and simplifying where we could to make things clearer.

Nivetha Kuruparan

1. *What went well while writing this deliverable?* Planning out the different modules early on was incredibly helpful for me. It allowed me to clearly identify how various parts of the system interact and what functionality could be combined or separated. This structured approach not only helped in designing the system but also made it easier to focus on what each module should accomplish, ensuring no major functionality was overlooked.

2. *What pain points did you experience during this deliverable, and how did you resolve them?* It was challenging for me to think through each module thoroughly and ensure that every input, output, and state variable was captured accurately. This required going through the implementation multiple times and considering edge cases that might not have been obvious at first. Breaking the process into smaller, more manageable tasks and carefully reviewing each module helped resolve this challenge.

Ayushi Amin

1. *What went well while writing this deliverable?* Honestly, once I got into it, things flowed pretty smoothly. Breaking everything down into smaller sections helped a ton. It made the whole thing feel less intimidating. I also felt like I had a good understanding

of how the modules all connected, which made it easier to explain things. We all had our own parts to work on based on the modules we have and were going to create so it was easier to work on something I was familiar with. Also, talking it through with my teammates about some of the trickier parts really helped me feel more confident about what I was writing. We all did code reviews and helped eachother out on parts we didn't quite get or thought we got. Overall, it felt pretty satisfying to see it all come together.

2. *What pain points did you experience during this deliverable, and how did you resolve them?* I think the hardest part of this was visualising extra dependencies and functions I would need to create to make my module work. We had coded out a portion of it but it did not include everything. I had to make sure I was not missing anything important. It felt like I was stuck in this loop of overthinking every little detail. To get past it, I took a break and came back with a fresh perspective, which helped a bit. I also hit up one of my teammates to talk through the parts I was struggling with. They gave me some ideas and helped me confirm I was on the right track since some of the modules I did were similar to theirs so we were able to collaborate easily. After that, things did not feel as stressful, and I was able to wrap it up.

Tanveer Brar

1. *What went well while writing this deliverable?* The best part about writing this deliverable was getting the chance to design the user interface before having implemented it. The Source Code Optimizer has already been designed and implemented as a result of the POC assignment in November. We had not implemented the VS Code Plugin for it yet, so getting the chance to actually think about its design was very rewarding(especially since most academic projects I have done before either involved no design component or very minimal for a small program). Each modules has clear responsibilities, which helped me anticipate all needed requirements for this plugin through a logical framework(POC implementation was a lot of trial and error). The other good thing were the built in labels for anticipated changes and modules, which helped me easily write down the traceability matrix.

2. *What pain points did you experience during this deliverable, and how did you resolve them?* One of the biggest challenges that I faced was identifying the correct module for each anticipated change in the traceability matrix. My team mate had worked on the anticipated changes, Some of these changes had overlapping responsibilities across modules, so I carefully reviewd the module responsibilities over again to be able to point out the modules for each change. It needed a lot of cross referencing the module guide and anticipated changes to make sure nothing was missde. Also, when determining module dependencies in the "Uses" section for each module's decomposition, I was not fully sure about which modules should depend on which for the VS Code Plugin. This is because there can be multiple possible ways, for example the Plugin Initializer or

Smell Detector being able to directly call Source Code Optimizer. While resolving this, I realized that while there is no one perfect mapping of dependencies, the goal should be to be as modular as possible and apply the seperation of concerns principle. This is why, for example, the Backend Communicator is the only module in the design that communicates with Source Code Optimizer.