Module Interface Specification for Software Engineering

Team 4, EcoOptimizers

Nivetha Kuruparan
Sevhena Walker
Tanveer Brar
Mya Hussain
Ayushi Amin

March 25, 2025

1 Revision History

Date	Version	Notes
January 17th, 2025	0.1	Initial Draft

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 analyze, refactor, and optimize 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]	a collection of objects of type T
set	set[T]	a collection of *unique* objects of type T
dictionary	dict	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
Software Decision Module	Measurements Module PylintAnalyzer Module Testing Functionality Module SmellRefactorer Module RefactorManager 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 Base Refactorer

`BaseRefactorer`

7.1 Module

The interface that all refactorers of this system will inherit from.

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, pylint_smell: dict, initial_emissions:` $\mathbb{R}$	None	None

7.4 Semantics

7.4.1 State Variables

- `temp_dir: Path`: Directory path for storing refactored files.

7.4.2 Environment Variables

None

7.4.3 Assumptions

- `output_dir` exists or can be created, and write permissions are available.

7.4.4 Access Routine Semantics

`BaseRefactorer(self, output_dir: Path)`

- **transition**: Initializes the `temp_dir` variable within `output_dir`.
- **output**: `self`
- **exception:** None.

`refactor(self, file_path: Path, pylint_smell: dict, initial_emissions: ℝ)`

- **transition**: Abstract method. No transition defined.
- **output**: None.
- **exception:** None.

7.4.5 Local Functions

None.

8 MIS of Long Message Chain Refactorer

`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
`LongMessageChainRefactorer`	`output_dir: Path`	`self`	None
`refactor`	`file_path: Path, pylint_smell: Smell, initial_emissions: ℝ`	None	`TypeError`, `IOError`

8.4 Semantics

8.4.1 State Variables

- **temp_dir**: Temporary directory for intermediate refactored files.

8.4.2 Environment Variables

- **File system**: Used to read, write, and store temporary and refactored files.
- **Logger**: Logs information during refactoring.

8.4.3 Assumptions

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

8.4.4 Access Routine Semantics

`LongMessageChainRefactorer(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: ℝ)`

- **Transition**:
  - Reads the file at `file_path`.
  - Identifies the line with a long message chain.
  - Refactors the chain by breaking it into separate statements.
  - Writes the refactored code to a temporary file.
  - Evaluates the refactored code's energy efficiency and functionality.
- **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.

8.4.5 Local Functions

`remove_unmatched_brackets(input_string: str)`

- **Transition**: Removes unmatched parentheses from the input string.
- **Output**: Returns the string with unmatched parentheses removed.
- **Exception**: None.

9 MIS of Long Lambda Function Refactorer

`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

- Uses `Smell` interface for data access
- Inherits from `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`, `pylint_smell: Smell`, `initial_emissions: ℝ`	None	`TypeError`, `IOError`

9.4 Semantics

9.4.1 State Variables

- **temp_dir**: Temporary directory for intermediate refactored files.

9.4.2 Environment Variables

- **File system**: Used to read, write, and store temporary and refactored files.
- **Logger**: Logs information during refactoring.

9.4.3 Assumptions

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

9.4.4 Access Routine Semantics

`LongLambdaFunctionRefactorer(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: ℝ)`

- **Transition**:
  - Reads the file at `file_path`.
  - Identifies the line with a long lambda function.
  - Refactors the lambda into a normal function.
  - 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.

9.4.5 Local Functions

`truncate_at_top_level_comma(body: str)`

- **Transition**: Truncates the lambda body at the first top-level comma, ignoring commas within nested parentheses, brackets, or braces.
- **Output**: Returns the truncated lambda body as a string.
- **Exception**: None.

10 MIS of Long Parameter List Refactorer

`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: ℝ`	None	`TypeError`, `IOError`

10.4 Semantics

10.4.1 State Variables

None

10.4.2 Environment Variables

- **File system**: Used for reading, writing, and storing temporary and refactored files.
- **Logger**: Logs details of the refactoring process.

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: ℝ)`

- **Transition**:
  1. Reads the file at `file_path` and locates the target function using `pylint_smell`.
  2. Analyzes function body to remove unused parameters. Updates the function signature and references in the code accordingly.
  3. If number of used parameters also exceeds the maximum configured limit, encapsulates related parameters into classes. Updates the function signature and references in the code accordingly.
  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: FuncDef, params: list[str]) -> set[str]`: Identifies parameters used within the function body.
2. `get_parameters_with_default_value(default_values: list[Constant], params: list[str]) -> dict`: Maps parameter names to their default values.
3. `classify_parameters(params: list[str]) -> dict`: Classifies parameters into `data` and `config` groups based on naming conventions.

4. `create_parameter_object_class(param_names: list[str], default_value_params: dict, class_name: str) -> str`: Generates class definitions for encapsulating parameters.

5. `update_function_signature(function_node: FuncDef, params: dict) -> FuncDef`: Updates function signatures to use encapsulated parameter objects.

6. `update_parameter_usages(function_node: FuncDef, params: dict) -> FuncDef`: Replaces parameter usages within the function body with attributes of encapsulated objects.

7. `update_function_calls(tree: Module, function_node: FuncDef, params: dict) -> Module`: Updates all calls to the refactored function.

11 MIS of Use List Accumulation Refactorer

`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 using Python's `join()` function, ensuring proper refactoring while maintaining the original functionality.

11.2 Uses

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

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: Real`	None	`TypeError`, `IOError`

11.4 Semantics

11.4.1 State Variables

- `target_line: int`: Line number where refactoring is applied.
- `target_node: ASTnode`: Node representing the concatenation variable.
- `assign_var: str`: Name of the variable the `target_node` represents.
- `last_assign_node: ASTnode`: Last initialization/assignment of the `assign_var` prior to the start of the loop.
- `concat_node: ASTnode`: Node where concatenation occurs.
- `scope_node: ASTnode`: Scope where refactoring is inserted.
- `outer_loop: ASTnode`: Outermost loop before the start of the concatenation.

11.4.2 Environment Variables

None

11.4.3 Assumptions

- The input file contains valid Python syntax.
- `pylint_smell` provides a valid line number for the detected code smell.

11.4.4 Access Routine Semantics

`UseListAccumulationRefactorer(self, output_dir: Path)`

- **transition**: Initializes the refactorer with `output_dir` and sets default state variables.
- **output**: `self`.
- **exception**: None

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

- **transition**: Parses `file_path`, identifies string concatenations in loops, modifies code for list accumulation, and writes refactored code to a file.
- **output**: None.
- **exception**: Raises `IOError` if input file cannot be read. Raises `TypeError` if source file cannot be parsed into an AST.

11.4.5 Local Functions

`find_last_assignment(self, scope: ASTnode)`

- **transition**: Identifies the last assignment of `assign_var` within the given `scope`.
- **output**: None.
- **exception**: Raises `TypeError` if given scope is null.

`find_scope()`

- **transition**: Finds the scope for refactoring based on AST node ancestry.
- **output**: None.
- **exception**: Raises `TypeError` if `concat_node` is not set.

`add_node_to_body(self, code_file: str)`

- **transition**: Inserts list accumulation and join statements into `code_file`.
- **output**: Returns the modified source code as a string.
- **exception**: Raises `TypeError` if `target_node` or `outer_loop` is not set.

12 MIS of Make Method Static Refactorer

`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

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

12.3 Syntax

Exported Constants: None
Exported Access Programs:

Name	In	Out	Exceptions
`MakeStaticRefactorer`	`output_dir: Path`	`self`	None
`refactor`	`file_path: Path`, `pylint_smell: Smell`, `initial_emissions: ℝ`	None	`TypeError`, `IOError`

12.4 Semantics

12.4.1 State Variables

- `target_line: int`: Line number where refactoring is applied.
- `mim_method_class: str`: Class name containing the method to refactor.
- `mim_method: str`: Method name to refactor.

12.4.2 Environment Variables

None

12.4.3 Assumptions

- The input file contains valid Python syntax.
- `pylint_smell` provides a valid line number for the detected code smell.

12.4.4 Access Routine Semantics

`MakeStaticRefactorer(self, output_dir: Path)`

- **transition**: Initializes the refactorer with `output_dir` and sets default state variables.
- **output**: `self`.
- **exception**: None.

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

- **transition**: Parses `file_path`, identifies the target function, modifies it to be static, and validates refactoring.
- **output**: None.
- **exception**: Raises `IOError` if input file cannot be read. Raises `TypeError` if source file cannot be parsed into an AST.

12.4.5 Local Functions

`visit_FunctionDef(self, node: FuncDef)`

- **transition**: Adds the `staticmethod` decorator to the target method and removes the `self` parameter if present.
- **output**: Returns the modified `FunctionDef` node.
- **exception**: None

`visit_ClassDef(self, node: ClassDef)`

- **transition**: Identifies the class containing the target method.
- **output**: Returns the modified `ClassDef` node.
- **exception**: None.

`visit_Call(self, node: Call)`

- **transition**: Updates method call references to use the class name instead of `self`.
- **output**: Returns the modified `Call` node.
- **exception**: None.

13 MIS of Long Element Chain Refactorer

`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

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

13.3 Syntax

13.3.1 Exported Constants

None

13.3.2 Exported Access Programs

Name	In	Out	Exceptions
`LongElementChainRefactorer`	`output_dir: Path`	`self`	None
`refactor`	`file_path: Path`, `pylint_smell: Smell`, `initial_emissions: ℝ`	None	`TypeError`, `IOError`

13.4 Semantics

13.4.1 State Variables

- **_reference_map**: Maps element chain references to their line numbers and corresponding values.

13.4.2 Environment Variables

- **File system**: Used to read, write, and store temporary and refactored files.
- **Logger**: Logs information during the refactoring process.

13.4.3 Assumptions

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

13.4.4 Access Routine Semantics

`LongElementChainRefactorer(output_dir: Path)`

- **Transition**: Initializes the refactorer with the specified output directory and sets up internal caching structures.
- **Output**: `self`.
- **Exception**: None.

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

- **Transition**:
  - Reads the file at `file_path`.
  - Identifies nested dictionary chains for flattening.
  - Refactors the identified chain by flattening the dictionary and replacing its occurrences.
  - 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.

13.4.5 Local Functions

- **_flatten_dict(d: dict[str, Any], parent_key?: str)**
  Recursively flattens a nested dictionary by combining keys with underscores.
- **_extract_dict_literal(node: ASTnode)**
  Converts an Abstract Syntax Tree (AST) dictionary literal into a Python dictionary.
- **_find_dict_assignments(tree: ASTnode, name: str)**
  Extracts dictionary assignments given the name of the dictionary from the AST and returns them as a dictionary.

- **_collect_dict_references(tree: ASTnode)**
  Identifies and stores all dictionary access patterns in the '_reference_map'.

- **_generate_flattened_access(base_var: str, access_chain: list[str])**
  Generates a flattened dictionary key string by combining elements of an access chain with underscores.

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 Python 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.

14.4.2 Environment Variables

- **TEMP**: Sets the temporary directory location for Windows systems. Used during the CodeCarbon energy measurement process.
- **TMPDIR**: Sets the temporary directory location for Unix-based systems. Used during the CodeCarbon energy measurement process.
- **Logger**: A logging mechanism that logs various events during the energy measurement process, including errors, completion of measurements, and other key actions.

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(file_path: Path)`

- **Transition**: Initializes the `CodeCarbonEnergyMeter` with the specified file path and logger. It sets up the necessary internal state for energy measurement and prepares the environment.
- **Output**: `self`.
- **Exception**: None.

`measure_energy()`

- **Transition**:
  - Logs the start of the energy measurement process.
  - Creates a temporary directory to store custom data.
  - Initializes the `EmissionsTracker` from CodeCarbon.
  - Runs the script specified by `file_path` and captures the output.
  - Stops the tracker after execution and stores the emissions data.
  - If available, it extracts the emissions data from the generated CSV file.
- **Output**:
  - Logs the results of the energy measurement process.
  - Stores the emissions data in `self.emissions_data`.
- **Exception**:
  - Logs an error if the file cannot be executed or if the emissions file is not created.
  - If the emissions data cannot be extracted from the CSV file, logs the issue.

14.4.5 Local Functions

`_extract_emissions_csv(csv_file_path: Path)` Extracts emissions data from a CSV file generated by CodeCarbon.

- **Input**: `csv_file_path` - The path to the CSV file containing emissions data.
- **Output**: Returns the last row of emissions data as a dictionary, or `None` if an error occurs.

15 MIS of Pylint Analyzer

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
__init__	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]: A list of detected code smells, represented as dictionaries.

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, such as extra Pylint options and custom smell definitions, are valid.

15.4.4 Access Routine Semantics

`__init__(self, file_path: Path, source_code: Module)`

- **transition**: Initializes the analyzer with the provided file path and AST of the source code.
- **output**: `self`.
- **exception:** None.

`build_pylint_options()`

- **transition**: Constructs the list of Pylint options based on the file path and configuration settings.
- **output**: Returns a list of strings representing Pylint options.
- **exception:** None.

`analyze()`

- **transition**: Executes Pylint analysis and custom checks, populating `smells_data` with detected smells.
- **output**: None.
- **exception:** Raises `JSONDecodeError` if Pylint's output cannot be parsed. Raises `Exception` for other runtime errors.

`configure_smells()`

- **transition**: Filters `smells_data` to include only configured smells.
- **output**: None.
- **exception:** None.

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

- **transition**: Filters the given Pylint results for a specific code smell identified by `code`.
- **output**: Returns a list of smells matching the specified code.
- **exception:** None.

15.4.5 Local Functions

`detect_long_message_chain(self, threshold?: int)`

- **transition**: Identifies method chains exceeding the specified `threshold`.
- **output**: Returns a list of smells for long method chains.
- **exception:** None.

`detect_long_lambda_expression(self, threshold_length?: int, threshold_count?: int)`

- **transition**: Detects lambda expressions exceeding length or expression count thresholds.
- **output**: Returns a list of smells for long lambda expressions.
- **exception:** None.

`detect_long_element_chain(self, threshold?: int)`

- **transition**: Detects dictionary access chains exceeding the specified `threshold`.
- **output**: Returns a list of smells for long dictionary chains.
- **exception:** None.

`detect_repeated_calls(self, threshold?: int)`

- **transition**: Identifies repeated function calls exceeding the `threshold`.
- **output**: Returns a list of smells for repeated function calls.
- **exception:** None.
- `parse_line(file_path: Path, line: int)`: Parses a specific line of code into an AST node.
- `get_lambda_code(lambda_node: Lambda)`: Returns the string representation of a lambda expression.

16 MIS of Testing Functionality

`TestRunner`

16.1 Module

Responsible for validating that any refactorings made to the source code do not modify it's original functionality.

16.2 Uses

- Uses Python's subprocess library

16.3 Syntax

Exported Constants: None
Exported Access Programs:

Name	In	Out	Exceptions
`__init__`	`run_command: str`, `project_path: Path`	`self`	None
`retained_functionality`	None	$\mathbb{B}$	`CalledProcessError`

16.4 Semantics

16.4.1 State Variables

- `project_path: Path`: Path to the source code directory.
- `run_command: str`: Command used to run the tests.

16.4.2 Environment Variables

None

16.4.3 Assumptions

- The provided `run_command` is a valid shell command.
- `project_path` is a valid path working source code directory.

16.4.4 Access Routine Semantics

`__init__(self, run_command: str, project_path: Path)`

- **transition**: Initializes the test runner with the given `run_command` and `project_path`.

- **output**: `self`.
- **exception**: None.

`retained_functionality()`

- **transition**: Runs the specified test command in the given project path. Logs success or failure, including standard output and error streams.
- **output**: Returns `True` if the tests passed; otherwise, returns `False`.
- **exception**: Raises a `CalledProcessError` if an eror occurs while running the tests in a subprocess.

16.4.5 Local Functions

None.

17 MIS of Use A Generator Refactorer

`UseAGeneratorRefactorer`

17.1 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.

17.2 Uses

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

17.3 Syntax

Exported Constants: None
Exported Access Programs:

Name	In	Out	Exceptions
`__init__`	`output_dir: Path`	`self`	None
`refactor`	`file_path: Path, pylint_smell: Smell, initial_emissions: ℝ`	None	`IOError`, `TypeError`

17.4 Semantics

17.4.1 State Variables

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

17.4.2 Environment Variables

None

17.4.3 Assumptions

- The input file contains valid Python syntax.
- `pylint_smell` provides a valid line number for the detected code smell.

17.4.4 Access Routine Semantics

`__init__(self, output_dir: Path)`

- **transition**: Initializes the `temp_dir` variable within `output_dir`.
- **output**: `self`.
- **exception:** None.

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

- **transition**: Parses `file_path`, identifies unnecessary list comprehensions, modifies the code to use generator expressions, and validates refactoring.
- **output**: None.
- **exception**: Raises `IOError` if input file cannot be read. Raises `TypeError` if source file cannot be parsed into an AST.

17.4.5 Local Functions

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

- **transition**: Replaces an `old_node` in the AST with a `new_node`.
- **output**: None.
- **exception**: None.

18 MIS of Cache Repeated Calls Refactorer

`CacheRepeatedCallsRefactorer`

18.1 Module

The `CacheRepeatedCallsRefactorer` module identifies repeated function calls in Python code and refactors them by caching the result of the first call to a temporary variable. This refactoring improves performance and energy efficiency while preserving the original functionality.

18.2 Uses

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

18.3 Syntax

Exported Constants: None
Exported Access Programs:

Name	In	Out	Exceptions
`__init__`	`output_dir: Path`	`self`	None
`refactor`	`file_path: Path, pylint_smell: Smell, initial_emissions: ℝ`	None	`IOError`, `TypeError`

18.4 Semantics

18.4.1 State Variables

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

18.4.2 Environment Variables

None

18.4.3 Assumptions

- The input file contains valid Python syntax.
- `pylint_smell` provides valid occurrences of repeated calls with line numbers and call strings.

18.4.4 Access Routine Semantics

`__init__(self, output_dir: Path)`

- **transition**: Initializes the `temp_dir` variable within `output_dir`.
- **output**: `self`.
- **exception:** None.

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

- **transition**: Parses `file_path`, identifies repeated function calls, inserts a cached variable for the first call, updates subsequent calls to use the cached variable, and validates refactoring.
- **output**: None.
- **exception**: Raises `IOError` if input file cannot be read. Raises `TypeError` if source file cannot be parsed into an AST.

18.4.5 Local Functions

- `_get_indentation(lines, line_number)`: Determines the indentation of a specific line.
- `_replace_call_in_line(line, call_string, cached_var_name)`: Replaces repeated calls with the cached variable.
- `_find_valid_parent(tree)`: Identifies the valid parent node containing all occurrences of the repeated call.
- `_find_insert_line(parent_node)`: Determines the line to insert the cached variable.

19 MIS of Plugin Initiator

19.1 Module

`Plugin Initiator` is a module that initializes the VS Code plugin and registers commands for VS Code Plugin.

19.2 Uses

- `Smell Detector` to register the command for detecting code smells.
- `Smell Refactorer` to register the command for refactoring user's selected code smell.

19.3 Syntax

Exported Constants: None
Exported Access Programs: None

19.4 Semantics

19.4.1 State Variables

None

19.4.2 Environment Variables

- **VS CODE API**: Used to register commands.

19.4.3 Assumptions

- The plugin is correctly loaded in VS Code.
- Source Code Optimizer executable is reachable and operational.

19.4.4 Access Routine Semantics

`activate()`

- **Transition**: Activates the plugin and registers commands.
- **Output**: None.
- **Exception**: None.

19.4.5 Local Functions

None

20 MIS of Backend Communicator

20.1 Module

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

20.2 Uses

Source Code Optimizer executable for energy measurement, smell detection and refactoring of applications.

20.3 Syntax

Exported Constants: None
Exported Access Programs:

Name	In	Out	Exceptions
`sendRequest`	`requestType: string, data: any`	Promise	Communication Error

20.4 Semantics

20.4.1 State Variables

None

20.4.2 Environment Variables

- **NETWORK**: Used for communicating with Source Code Optimizer.
- **LOGGER**: Logs any communication errors.

20.4.3 Assumptions

- Source Code Optimizer executable is reachable and operational.
- Python file with no syntax errors is present in the VS Code editor.

20.4.4 Access Routine Semantics

`sendRequest(requestType: str, data: Any)`

- **Transition**: Sends the provided request to Source Code Optimizer and receives a response.
- **Output**: A promise that resolves with Source Code Optimizer's response.
- **Exception**: Logs any errors encountered during communication.

20.4.5 Local Functions

None

21 MIS of Smell Detector

21.1 Module

`Smell Detector` analyzes the active file for code smells and interacts with Source Code Optimizer for detection.

21.2 Uses

- `Backend Communicator` for communicating with Source Code Optimizer for smell detection.
- `File Highlighter` for highlighting detected smells in the editor.

21.3 Syntax

Exported Constants: None

Exported Access Programs:

Name	In	Out	Exceptions
`detect`	None	None	Active file not found

21.4 Semantics

21.4.1 State Variables

None

21.4.2 Environment Variables

- **EDITOR**: Used to access the active file.

21.4.3 Assumptions

- There is an active Python file with no syntax error in the editor.
- Source Code Optimizer correctly identifies smells.

21.4.4 Access Routine Semantics

`detect()`

- **Transition**: Reads the active file, sends it to Source Code Optimizer for analysis, and highlights detected smells in the editor.
- **Output**: None.
- **Exception**: Throws an error if no active file is found.

21.4.5 Local Functions

None

22 MIS of Smell Refactorer

22.1 Module

`Smell Refactorer` applies a refactoring to a detected smell.

22.2 Uses

- `Backend Communicator` for sending the smell data to Source Code Optimizer for refactoring.
- `Refactor Manager` for managing refactoring workflows.

22.3 Syntax

Exported Constants: None

Exported Access Programs:

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

22.4 Semantics

22.4.1 State Variables

None

22.4.2 Environment Variables

- **EDITOR**: Used to apply refactored changes.

22.4.3 Assumptions

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

22.4.4 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.

22.4.5 Local Functions

None

23 MIS of File Highlighter

23.1 Module

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

23.2 Uses

None

23.3 Syntax

Exported Constants: None
Exported Access Programs:

Name	In	Out	Exception
`highlight`	`range: range[]`	None	None
`clear`	`None`	None	None

23.4 Semantics

23.4.1 State Variables

- `highlightedRanges`: Stores the currently highlighted regions in the editor.

23.4.2 Environment Variables

- **EDITOR**: Used to apply and clear highlights.

23.4.3 Assumptions

- The VS Code editor is active and accessible.
- Python file with no syntax errors is currently open in the editor.

23.4.4 Access Routine Semantics

`highlight(ranges: Range[])`

- **Transition**: Adds highlights to the specified ranges in the editor.
- **Output**: None.
- **Exception**: None.

`clear()`

- **Transition**: Removes all highlights from the editor.
- **Output**: None.
- **Exception**: None.

23.4.5 Local Functions

None

24 MIS of Hover Manager

24.1 Module

`Hover Manager` manages hover effects to display contextual information.

24.2 Uses

- `File Highlighter` for providing contextual information about highlighted smells.

24.3 Syntax

Exported Constants: None

Exported Access Programs:

Name	In	Out	Exception
`showHover`	`position: Position`	None	None
`clearHover`	`None`	None	None

24.4 Semantics

24.4.1 State Variables

- `currentHover`: Stores the currently displayed hover information.

24.4.2 Environment Variables

- **EDITOR**: Used to display hover effects.

24.4.3 Assumptions

None

24.4.4 Access Routine Semantics

`showHover(position: Position)`

- **Transition**: Displays hover information at the specified position.
- **Output**: None.
- **Exception**: None.

`clearHover()`

- **Transition**: Clears any active hover information.
- **Output**: None.
- **Exception**: None.

24.4.5 Local Functions

None

25 MIS of Refactor Manager

25.1 Module

`Refactor Manager` manages the process of applying refactorings to detected smells.

25.2 Uses

None

25.3 Syntax

Exported Constants: None

Exported Access Programs:

Name	In	Out	Exceptions
`applyRefactor`	`refactor: Refactor`	None	Validation error
`previewRefactor`	`refactor: Refactor`	None	None
`undoRefactor`	None	None	None

25.4 Semantics

25.4.1 State Variables

- `appliedRefactors`: Stores a history of applied refactors.

25.4.2 Environment Variables

- **EDITOR**: Used to apply and preview refactors.

25.4.3 Assumptions

- The refactoring data is valid and corresponds to detected smells.

25.4.4 Access Routine Semantics

`applyRefactor(refactor: Refactor)`

- **Transition**: Applies the provided refactor to the active editor.
- **Output**: None.
- **Exception**: Logs validation errors if the refactor cannot be applied.

`previewRefactor(refactor: Refactor)`

- **Transition**: Displays a preview of the refactor in the editor.

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

`undoRefactor()`

- **Transition**: Reverts the most recently applied refactor.
- **Output**: None.
- **Exception**: None.

25.4.5 Local Functions

None

26 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 CO2 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 CO2 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 visualizing 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.