**Masters of Applied Computer Science**

Name **: Dhruv Hiteshbhai Upadhyay**

Registration Number **: B00947866**

Professor/Supervisor **: Dr. Tushar Sharma**

**CSCI 9301 & 9302**

**Python Smell Detector Tool**

1. **Smell Detector Testing**:
   1. Tool already was at the stage of Implementation and Design smells detection.
   2. Testing of the tool for both the smells against manual detection was the first step.
   3. This step involved following pathway:
      1. Selection of three organization based projects namely “Acoular”, “Noisemaker”, and “Phidl”. These projects were selected with more than 10,000 lines of code, more than 100 stars and more than 100 commits of advance search.
      2. Manual detection of 15 Design smells and 9 Implementation smells was carried out.
      3. Then the detection of tool and manual detection were compared for all three projects.
      4. This lead to deeper understanding of smells and issues creation.
   4. Due to this testing process I detected issues in *Long Statement smell*, *Long Parameter smell*, and *Missing Default smell*.
   5. The Log Statement smell had an issue in conditional and looping statements. As the tool would consider all the lines of code under those statements as on single line of code and not multi line structure.
   6. The Long Parameter smell had an issue in falsely detecting self as parameter and hence at times crossing the threshold to state the smell.
2. **Existing Issues Solution**:

After finding the issues, I created issues in Github and started working on them one by one. Understanding the project structure and AST (Abstract Syntax Tree) working to retrieve Packages, Modules, Classes, Functions and Methods were core part of learning. After understanding the entire code structure and working, I solved the issues that were pertinent in Implementation and Design smells.

1. **Architecture Smells**:

Introduction:

Architecture smells are design issues or deficiencies in the structure of a software system that can lead to various problems such as decreased maintainability, reduced scalability, and increased complexity. These smells indicate deviations from established architectural principles and best practices, potentially impacting the system's quality attributes [2].

List of Seven Types of Architecture Smells:

* 1. Ambiguous Interface
  2. Cyclic Dependency
  3. God Component
  4. Feature Concentration
  5. Dense Structure
  6. Scattered Functionality
  7. Unstable Dependency

Each type of architecture smell represents a specific pattern of design or implementation issue that can degrade the overall quality and maintainability of a software system. Identifying and addressing these smells early in the development process is crucial for ensuring the long-term viability and sustainability of the system.

1. **Ambiguous Interface**:

Ambiguous Interfaces are interfaces that offer only a single, general entry-point into a component [2]. This smell appears especially in event-based publish-subscribe systems, where interactions are not explicitly modeled and multiple components exchange event messages via a shared event bus [2].

Detection Mechanism:

* 1. Iterate through all modules within each package [2].
  2. For each module, iterate through its functions and methods [2].
  3. Check if any method or function serves as a single, general entry point (e.g., marked as public) [2].
  4. If found, flag it as a potential instance of the Ambiguous Interface smell [2].

Implementation Steps:

1. Iterate through each package and its modules.
2. For each module, iterate through its functions and methods.
3. Check if the access modifier of each method/function is "public," indicating it as an entry point.
4. If a method/function with a "public" access modifier is found, flag it as a potential instance of the Ambiguous Interface smell.
5. Aggregate and return the detected smells for each package.

Pseudocode:

function detect\_ambiguous\_interface\_smells(modules, config, package\_name):

Initialize an empty list to store detected smells.

Iterate through each module in the package:

Iterate through each function/method in the module:

If the function/method has a public access modifier:

Flag it as a potential instance of the Ambiguous Interface smell.

Return the list of detected smells.

function detect\_smells(package\_details, config):

Initialize an empty list to store all detected smells.

Iterate through each package in the project:

Detect ambiguous interface smells in the package.

Append the detected smells to the list of all smells.

Return the list of all detected smells.

Conclusion:

The implemented logic iterates through each module, function, and method to identify public entry points that could lead to ambiguity in interface usage. Detected instances of the Ambiguous Interface smell are logged and returned for further analysis and resolution.

1. **Cyclic Dependency**:

This smell occurs when two or more packages/subsystems depend on each other. Cycles between packages/subsystems can be created through use, inheritance, or through a combination of use and inheritance [1].

Detection Mechanism:

* 1. Construct a dependency graph representing the dependencies among packages [2].
  2. Use depth-first search (DFS) to detect cycles in the dependency graph.
  3. If a cycle is found, flag it as a potential instance of the Cyclic Dependency smell [2].

Implementation Steps:

1. Construct a dependency graph using the provided package details and their dependencies.
2. Use DFS to traverse the dependency graph and detect cycles.
3. If a cycle is found, log it as a detected smell along with relevant details.
4. Aggregate and return all detected smells.

Pseudocode:

function find\_cycles(graph):

Initialize an empty set to store visited nodes.

Initialize an empty list to store detected cycles.

Define a recursive function dfs(node, parent, path):

Mark the current node as visited.

Append the current node to the path.

Iterate through neighbors of the current node:

If a neighbor is not visited, recursively call dfs on it.

If a neighbor is visited and not the parent, a cycle is detected.

Add the cycle to the list of detected cycles.

Remove the current node from the path.

Iterate through nodes in the graph:

If a node is not visited, call dfs on it.

Return the list of detected cycles.

function detect\_smells(package\_details, config):

Initialize an empty list to store all detected smells.

Construct a dependency graph from the package details.

Detect cycles in the dependency graph using the find\_cycles function.

Log detected cycles as instances of the Cyclic Dependency smell.

Aggregate and return all detected smells.

Conclusion:

The implemented logic constructs a dependency graph from package details and applies DFS to detect cycles. Detected instances of the Cyclic Dependency smell are logged and returned for further analysis and resolution.

1. **God Component**:

This smell occurs when a component is excessively large either in terms of Lines of Code (LOC) or number of classes [2].

Detection Mechanism:

* 1. Calculate the Lines of Code (LOC) and the number of classes in each package/module [2].
  2. Define thresholds for acceptable LOC and class count [2].
  3. Flag a package/module as a potential instance of the God Component smell if it exceeds the predefined thresholds [2].

Implementation Steps:

1. Iterate through each package/module in the system.
2. Calculate the total Lines of Code (LOC) and the total number of classes within each package/module.
3. Compare the total LOC and class count against predefined thresholds.
4. If the total LOC or class count exceeds the thresholds, log the package/module as a detected smell.
5. Aggregate and return all detected smells.

Pseudocode:

function detect\_smells(package\_details, config):

Initialize an empty list to store all detected smells.

Define thresholds for acceptable LOC and class count.

Iterate through each package/module in the system:

Calculate the total Lines of Code (LOC) and the total number of classes.

If the total LOC or class count exceeds the predefined thresholds:

Log the package/module as a potential instance of the God Component smell.

Aggregate and return all detected smells.

Conclusion:

The implemented logic iterates through each package/module, calculates the total LOC and class count, and compares them against predefined thresholds. Detected instances of the God Component smell are logged and returned for further analysis and resolution.

1. **Feature Concentration**:

A component suffers from the feature concentration architecture smell when one architectural component realizes more than one architecture concern [1].

Detection Mechanism:

* 1. Calculate the Lack of Component Cohesion (LCC) metric, representing the ratio of disconnected subgraphs to the total number of classes in a package/module [2].
  2. Define a threshold for acceptable LCC [2].
  3. Flag a package/module as a potential instance of the Feature Concentration smell if its LCC exceeds the predefined threshold [2].

Implementation Steps:

* 1. Iterate through each package/module in the system.
  2. Build a dependency graph for each module to represent class-level dependencies.
  3. Count the number of disconnected subgraphs in the dependency graph.
  4. Calculate the Lack of Component Cohesion (LCC) by dividing the total disconnected subgraphs by the total number of classes.
  5. If the LCC exceeds the predefined threshold, log the package/module as a detected smell.
  6. Aggregate and return all detected smells.

Pseudocode:

function detect\_smells(package\_details, config):

Initialize an empty list to store all detected smells.

Define a threshold for acceptable Lack of Component Cohesion (LCC).

Iterate through each package/module in the system:

Build a dependency graph for the module.

Count the number of disconnected subgraphs in the dependency graph.

Calculate the Lack of Component Cohesion (LCC).

If the LCC exceeds the predefined threshold:

Log the package/module as a potential instance of the Feature Concentration smell.

Aggregate and return all detected smells.

Conclusion:

The implemented logic iterates through each package/module, builds a dependency graph, counts the disconnected subgraphs, calculates the Lack of Component Cohesion (LCC), and flags instances of the Feature Concentration smell if the LCC exceeds the predefined threshold. Detected smells are logged and returned for further analysis and resolution.

1. **Dense Structure**:

This smell arises when components have excessive and dense dependencies without any particular structure [2].

Detection Mechanism:

* 1. Build a dependency graph representing the architecture, where nodes represent components (e.g., classes, modules) and edges represent dependencies between them [2].
  2. Calculate the average degree of the dependency graph, representing the average number of dependencies per component [2].
  3. Define a threshold for acceptable average degree [2].
  4. Flag the architecture as exhibiting the Dense Structure smell if the average degree exceeds the predefined threshold [2].

Implementation Steps:

1. Iterate through each package/module in the system.
2. Build a dependency graph for the entire architecture based on class-level dependencies.
3. Compute the average degree of the dependency graph.
4. If the average degree exceeds the predefined threshold, log the architecture as a potential instance of the Dense Structure smell.
5. Aggregate and return all detected smells.

Pseudocode:

function detect\_smells(package\_details, config):

Initialize an empty list to store all detected smells.

Define a threshold for acceptable average degree.

Build a dependency graph for the entire architecture.

Calculate the average degree of the dependency graph.

If the average degree exceeds the predefined threshold:

Log the architecture as exhibiting the Dense Structure smell.

Aggregate and return all detected smells.

Conclusion:

The implemented logic iterates through each package/module, builds a dependency graph for the entire architecture, calculates the average degree, and flags instances of the Dense Structure smell if the average degree exceeds the predefined threshold. Detected smells are logged and returned for further analysis and resolution.

1. **Scattered Functionality**:

Scattered Functionality describes a system where multiple components are responsible for realizing the same high-level concern [1].

Detection Mechanism:

* 1. Identify methods that access multiple components [2].
  2. Check if accesses to at least two external components occur from a method [2].
  3. Flag the method as exhibiting the Scattered Functionality smell if multiple components are accessed [2].

Implementation Steps:

1. Iterate through each package/module in the system.
2. Traverse each class and method within the module.
3. For each method, determine the components accessed.
4. If the method accesses multiple components (excluding components within the same package), flag it as exhibiting the Scattered Functionality smell.
5. Aggregate and return all detected smells.

Pseudocode:

function detect\_smells(package\_details, config):

Initialize an empty list to store all detected smells.

For each package/module in the system:

For each class in the module:

For each method in the class:

Determine the components accessed by the method.

If the method accesses multiple external components:

Log the method as exhibiting the Scattered Functionality smell.

Aggregate and return all detected smells.

Conclusion:

The implemented logic iterates through each method in the system, identifies the components accessed, and flags instances of the Scattered Functionality smell if a method accesses multiple external components. Detected smells are logged and returned for further analysis and resolution.

1. **Unstable Dependency**:

Unstable dependency architecture smell arises in a component when the component depends on a less stable component [1][2].

Detection Mechanism:

* 1. Identify the incoming and outgoing dependencies for each package/module in the system [2].
  2. Calculate the instability ratio for each package using the formula: outgoing dependencies / (outgoing dependencies + incoming dependencies) [2].
  3. Flag packages with an instability ratio exceeding a predefined threshold as exhibiting the Unstable Dependency smell [2].

Implementation Steps:

1. Iterate through each package/module in the system.
2. Determine the incoming and outgoing dependencies for each package/module.
3. Calculate the instability ratio for each package/module.
4. Compare the instability ratio with the predefined threshold.
5. Flag packages with an instability ratio exceeding the threshold as exhibiting the Unstable Dependency smell.
6. Aggregate and return all detected smells.

Pseudocode:

function detect\_smells(package\_details, config):

Initialize an empty list to store all detected smells.

For each package/module in the system:

Calculate the incoming and outgoing dependencies for the package/module.

Calculate the instability ratio using the formula.

If the instability ratio exceeds the threshold:

Log the package/module as exhibiting the Unstable Dependency smell.

Aggregate and return all detected smells.

Conclusion:

The implemented logic iterates through each package/module in the system, calculates the instability ratio based on incoming and outgoing dependencies, and flags packages/modules with instability ratios exceeding the predefined threshold as exhibiting the Unstable Dependency smell. Detected smells are logged and returned for further analysis and resolution.

1. **Demo Screen Shots**:

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

References:

[1] “A Taxonomy of Software Smells,” [*www.tusharma.in*](http://www.tusharma.in). <https://www.tusharma.in/smells/ARCH.html> (accessed Apr. 05, 2024).

[2] *Tushar Sharma, Paramvir Singh, Diomidis Spinellis*, “An Empirical Investigation on the Relationship between Design and Architecture Smells,” (accessed Apr. 05, 2024).