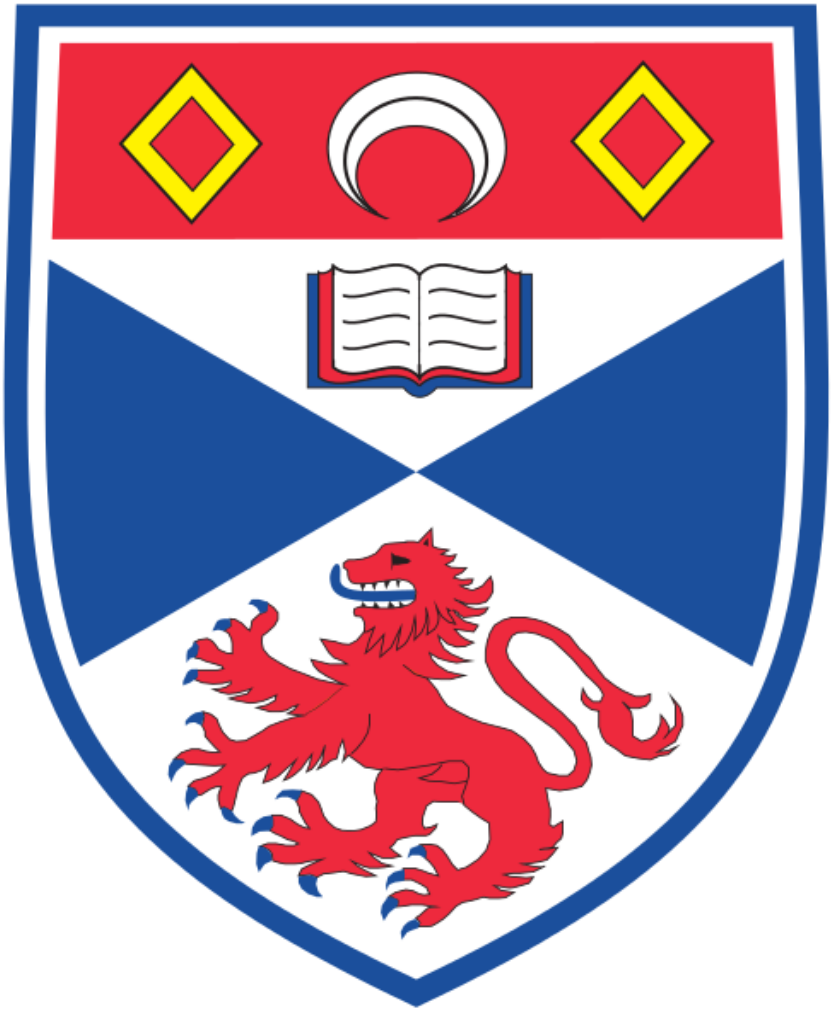
Inspect4j – A Comprehensive Static Code Analysis for Java Software Repositories

Nandi Mbimbi Ncube

190031593



Project Supervisor: Dr Rosa Filgueira

School of Computer Science
University of St Andrews
22nd March 2024

Abstract

In academic research, software adoption is key to promoting scientific knowledge discovery and innovation. However, successful software adoption heavily depends on the comprehension of code. Without a clear understanding of software, the ability to critically assess, select, and effectively use software is compromised. To enrich understanding and facilitate easier code comprehension, this project proposes a new static code analysis framework called Inspect4J, which provides various insights about Java code repositories. The framework follows in the footsteps of Inspect4py, an existing framework which extensively analyses Python code bases. When provided with a repository, Inspect4J extracts and presents information regarding its classes, interfaces, methods, dependencies and required libraries. The application has undergone comprehensive testing and evaluation, and its limitations have been outlined in the report. Overall, this project contributes to the discipline of state code analysis by introducing a new framework which enhances code understandability and streamlines software integration for researchers and developers.

Declaration

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated.

The main text of this project report is 9,911 words long, including project specification and plan.

In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

Table of Contents

Table of Listings

Table of Figures

Table of Tables

1 Introduction

Recently, there has been increased recognition and promotion of the production of Findable, Accessible, Interoperable, and Reusable (FAIR) scholarly digital assets and data (Wilkinson, et al., 2016). The current lack in prioritisation of good data management and stewardship has led to the minimisation of knowledge discovery, innovation, evaluation and reuse of academic software, algorithms, tools, and workflows (Wilkinson, et al., 2016). As reported by Filgueira and Garijo (2022), while the adoption of FAIR principles in an attempt to promote these aspects is on the increase, software adoption and reuse continues to be a laborious process.

Filgueira and Garijo (2022) present Inspect4py, a static code analysis framework which strives to make adoption of Python software more efficient and streamlined. The framework is an open-source application that extensively analyses Python code repositories (Filgueira & Garijo, 2022). Inspect4py aims to provide users with an overview of the primary software features of the Python source code and an estimation of the type of software contained in the repository. By offering these insights, this framework enables others to form a deeper understanding of code repositories and assess their suitability for integration and reuse (Filgueira & Garijo, 2022). Taking a leaf from Filgueira and Garijo (2022), this report presents Inspect4j, a static code analysis framework that aims to broaden the scope of Inspect4py by allowing for the analysis of Java code repositories.

This report has been divided into eight parts. The remainder of the first part presents the framework's objectives and the contributions made. The second part explores the background of static code analysis and introduces Abstract Syntax Trees, the data structure which underpins analysis in this domain. It also investigates Inspect4py and other related frameworks. The third part presents the project's requirements specification. The fourth part outlines the developmental approach taken and defends its usage, which is then followed by a short discussion of the project's ethics in the fifth part. The sixth part gives an in-depth description of the framework's architectural design and an explanation of the framework's implementation. The seventh part critically evaluates the application against its objective. Finally, concluding with the eighth part, which gives a summary of the project's key contributions and limitations and discusses areas for potential future work.

1.1 Objectives

This project seeks to produce a new framework which extends the capabilities of the Inspect4py framework to the Java domain, with the intention of addressing the following aspects:

1. Implementation of a minimum viable product (MVP) which operates at the file level, allowing for basic analysis of classes, interfaces, and methods.
2. Provide additional functionality by allowing for retrieval of higher-level, "context-aware" insights based on the information gathered about the repository in its entirety.

These aspects have been captured in the following objectives. The objectives presented have been adapted from the Description, Objectives, Ethics and Resource document devised in the early stages of this project. The details of the following objectives are expanded and assessed in Parts 6 and 7.

Primary

1. Gain familiarity and an understanding of the type of insights provided by Inspect4py and the underpinning concepts and practices that enable information extraction.
2. Gain familiarity with software that builds Abstract Syntax Trees (ASTs) from given Java source code.
3. Implement a minimum viable product that performs class, interface, and method analysis. This product should perform the following:
    - Extraction of class, interface, and method metadata, including retrieval of information about inheritance relationships, nested classes, and nested interfaces.
    - Capturing of method signatures, parameters, return values, and relevant variables.
    - Identification and analysis of other method types, such as lambda expressions and method references.
    - Retrieval of class, interface, and method documentation.

Secondary

1. Extend the minimum viable product by adding a feature that extracts file metadata, such as the presence of a main method.
2. Implement a feature which identifies and records the organisation and grouping of files within the repository.
3. Implement a feature which identifies repository requirements and dependencies. The component should address the following:
    - Must be able to amass a list of required Java versions and other packages and their versions.
    - Must be able to identify internal and external classes and interfaces utilised within the code base. This includes the detection of top-level and nested/inner classes local to the software, as well as classes and interfaces from the Java Standard Library and external packages.

Tertiary

1. Implement a feature that can identify files dedicated to software testing.

1.2 Contributions

The application produced in this project is a static code analysis framework derived from Inspect4py, which should provide users with an outline of key software features at varying levels of abstraction for each file within a single repository. An initial set of objectives was developed in the early stages of the project, which were presented in Section 1.1. After completion of the project, these objectives, and the extent to which they were addressed were assessed. The results of the evaluation are presented in Part 7.

- Primary objective 1 was achieved and discussed in Part 6.
- Primary objective 2 was achieved and discussed in Part 6.
- Primary objective 3 was achieved and discussed in Part 6.
- Secondary objective 1 was achieved and discussed in Part 6.
- Secondary objective 2 was achieved and discussed in Part 6.
- Secondary objective 3 was achieved and discussed in Part 6.
- Tertiary objective 1 is incomplete and discussed in Part 7.

2 Context Survey

2.1 Background

The purpose of this section is to provide contextual and background information about static code analysis and its significance. Furthermore, it describes how static analysers use Abstract Syntax Trees (AST) to facilitate their data extraction.

2.1.1 Static Code Analysis and its Frameworks

Static Code Analysis refers to the inspection of program source code without executing it (Elkhalifa & Ilyas, 2016). In recent years, static code analysis frameworks and tools have become increasing important during software development (Novak, et al., 2010). Typically, they are used as a software verification technique, employed with the objective of unveiling potential issues within the code (Elkhalifa & Ilyas, 2016). Identification and removal of such issues plays a crucial role in ensuring that software is safe, secure, reliable, and maintainable (MathWorks, n.d.). Static code analysis tools and frameworks come in many flavours and usually differ in the set of issues they can identify. However, commonly targeted issues include, syntactic errors, potential security vulnerabilities, poor coding practices, concurrency issues, violations of coding standards and memory leaks (Novak, et al., 2010, Ayewah, et al., 2008 & Chaudhary, 2022). While static code analysis tools are extensively used for software verification, tools such as Inspect4py offer functionalities that extend beyond this scope.

2.1.2 Abstract Syntax Trees

The Cornell University Department of Computer Science (2022) refers to an Abstract Syntax Tree (AST) as a high-level hierarchical tree structure that defines source code as a set of nodes, where each node denotes a concept in a given programming language. These nodes are linked by edges, serving to symbolise the relationships among the various programming constructs. An example is provided in Listing 1 and Figure 1 to illustrate this concept further.

ASTs have been a valuable data structure in numerous aspects of software development, notably in program compilation. During compilation, ASTs function as intermediate representation produced after tokenisation and parsing of source code. In the lexical analysis phase of compilation, the source code is the decomposed into tokens, where each token represents a construct within the programming language in which the source code is written (Finamore, 2021). For example, a token may represent an operator, variable or literal (Finamore, 2021). Subsequently, during the Parsing phase, these tokens are connected to build the AST (Finamore, 2021). Within the compilation process, ASTs are a useful representation of code because they exclude minor syntactic elements (e.g., the presence of semi-colons), which allows for future optimisation on source code to be performed on those parts relevant to the compiler (Racordon, 2021 & Stratego/XT, n.d.).

As previously noted, ASTs serve many purposes, purposes that go beyond program compilation. Schiewe et al. (2022) states that static code analysis approaches traditionally depend on Abstract Syntax Trees (ASTs) to retrieve information about software. Tools such as ESLint and SonarQube leverage this underling tree structure to detect problematic patterns or issues such as poor coding practices and potential security vulnerabilities (ESLint, n.d. & SonarQube, n.d.). While beneficial for identifying issues, frameworks such as Inspect4py diverge from this, instead using ASTs to generate concise summaries of program source code.

```
if a > b:
    b := a - b
else:
    b := a * a
```

Listing 1: A pseudo-code snippet of an if/else statement

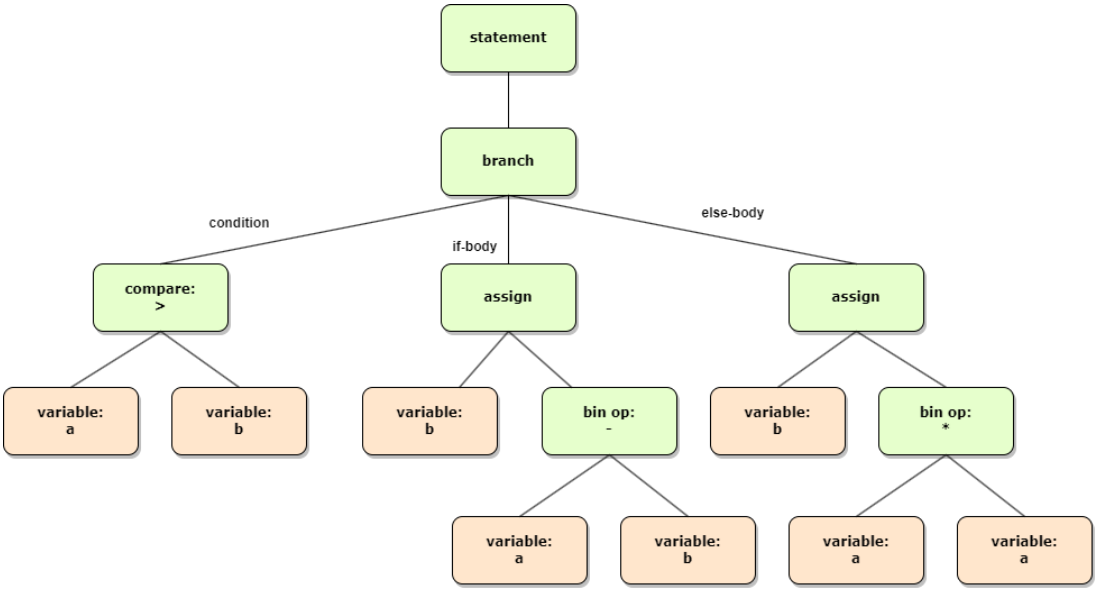


Figure 1: The AST for the example if/else statement.

2.1.3 Inspect4py

Inspect4py is a static code analysis framework, proposed by researchers Dr Rosa Filgueira and Dr Daniel Garijo, which aims to extract comprehensive information about Python code repositories (Filgueira & Garijo, 2022). When provided with a repository, the framework is designed to offer insights on two main aspects.

1) **Software understanding features, which includes:**
    - Class and function metadata and documentation: Inspect4py provides comprehensive insights into class and function metadata, encompassing various attributes. As stated by Filgueira & Garijo (2022), for each class detected in a repository, the framework reports back various details, such as its name, inherited classes, documentation, and methods. Similarly, for each function, it retrieves information such as its arguments, documentation, return values, variables which store the results of other function calls and nested functions.
    - Dependencies and Requirements: Inspect4py compiles and presents a list of a packages (including their corresponding version) necessary for the execution of the target software (Filgueira & Garijo, 2022 & Filgueira & Williams, 2023).
    - Tests: Inspect4py identifies and returns a list of files intended for testing the functional components of the target software (Filgueira & Garijo, 2022).
    - Main Software Type and Software Invocation: Inspect4py provides an approximation of the software type of the repository and alternative software invocation procedures. More specifically, the tool categorises the software as a package, library, service, or a set of scripts and, based on this classification, provides a ranked list of alternative ways to execute it (Filgueira & Garijo, 2022).

2) **Code features, which includes:**
    - File metadata: Inspect4py captures and stores information about each file. This includes data about the internal and external modules it imports and whether the file contains a main method or a file body (Filgueira & Garijo, 2022).
    - Control Flow graph: For each file, Inspect4py generates a textual and graphical representation (stored in a text file or in a png/dot/pdf file) of the software's execution flow and paths (Filgueira & Garijo, 2022).
    - Call graph: For each code block, such as a function or method, Inpect4py extracts a call graph of all involved functions (Filgueira & Garijo, 2022).
    - File hierarchy: The framework documents how the files have been grouped into directories and how these directories are organised within the repository (Filgueira & Garijo, 2022).

An illustration of how Inspect4py presents its insights is demonstrated in Listing 2 and Listing 3. To gather the information seen in Listing 3, inspect4py converts the source code seen in Listing 2 to an AST and then "walks" through the tree, stopping at relevant nodes and extracting information from them. The information is subsequently structured, formatted, and written to a JSON or HTML file, enabling users to conveniently view the information.

```python
import random
def generate_random_list(length, min_value, max_value):
    return [random.randint(min_value, max_value) for _ in range(length)]

def main():
    length = random.randint(5, 15)
    min_value = 2
    max_value = 5

    random_list = generate_random_list(length, min_value, max_value)
    print("Generated Random List:", random_list)

if __name__ == "__main__":
    main()
```

Listing 2 An example python file supplied to inspect4py.

The primary objective of inspect4py is to enhance the efficiency of software reuse and code adoption. The insights provided by Inspect4py allow developers to gain a deeper understanding of unseen code bases, which facilitates smoother integration with other software. However, its primary limitation is that it only supports Python code bases. Although Python is widely used, numerous individuals and organisations rely on software programmed in other languages such as Java. Therefore, highlighting the necessity for an alternative framework, which aims to deliver the same services to those within the Java domain.

```
{
    "file": {
        "path": "C:\\Users\\nm\\OneDrive\\Documents\\ex\\cli.py",
        "fileNameBase": "cli",
        "extension": "py"
    },
    "dependencies": [
        {
            "import": "random",
            "type": "external",
            "type_element": "module"
        }
    ],
    "functions": {
        "main": {
            "min_max_lineno": {
                "min_lineno": 3,
                "max_lineno": 6
            },
            "calls": [
                "random.randint",
                "print"
            ],
            "store_vars_calls": {
                "rand_num": "random.randint"
            }
        }
    },
    "body": {
        "calls": [
            "cli.main"
        ]
    },
    "main_info": {
        "main_flag": 1,
        "main_function": "cli.main",
        "type": "script"
    },
    "is_test": false
}
```


Listing 3: The corresponding JSON file outputted by inspect4py containing the information it extracts about the example python file.

2.1.3.1 *RepoPyOnto Ontology*

In the paper titled “RepoGraph: A Novel Semantic Code Exploration Tool for Python Repositories Based on Knowledge Graphs and Deep Learning.”, Filgueira and Williams (2023) propose the RepoPyOnto ontology (Table 1 and Table 2). This ontology formally describes the main entities and relationships observed in Python source code, and covers the key entities identified by inspect4py (Filgueira & Williams, 2023) .

Entity	Properties
Repository	name, full_name, description, type, ...
README	path, content
License	text,license_type, confidence
Directory	name, path, parent_path
Package	name, canonical_name, parent_package, path, parent_path, external, inferred
Module	name, canonical_name, path, parent_path, extension, is_test, inferred
Class	name, canonical_name, min_line_number, max_line_number
Function	name, type, builtin, canonical_name, source_code, ast, min_line_number, max_line_number, inferred
Variable	name, canonical_name, type, inferred
Argument	name, type
Return Value	name, type
Body	source_code
Docstring	short_description, long_description, summarization
Docstring Argument	name, type, description, is_optional, default
Docstring Return Value	name,type, description, is_generator
Docstring Raises	description, type

Table 1: RepoPyOnto entity set (Filgueira & Williams, 2023)

Relationship	Source	Destination
Requires	Repository	Package
Contains	Repository, Directory, Package, Module	Directory, Module, Package, README, Function, Class, Body, Variable
Imports	Module	Module, Package, Class, Function, Variable
HasMethod	Class	Function
HasFunction	Module	Function
Extends	Class	Class
HasArgument	Function	Argument
LicensedBy	Repository	License
Documents	Docstring	Function, Class
Describes	Docstring	DocstringArgument, DocstringRaises, DocstringReturnValue
Calls	Module, Function	Function, Class, Module

Table 2 : RepoPyOnto relationship set (Filgueira & Williams, 2023)

As depicted in Table 1 and Table 2, each entity type has a set of properties that distinguish instances of that type. These instances, where appropriate, may establish relationships amongst themselves which can be described using the relationship descriptor presented in Table 2. For instance, a Class entity with the name “Student” may serve as a subclass of the Class entity with the name “Person”. Consequently, they share an “Extends” relationship, where the source is the “Student” Class entity, and the destination is the “Person” Class entity.

Inspect4J endeavours to closely align itself with this ontology to ensure that there is consistency between itself and Inspect4py. Section 6.1.1 discusses how this ontology is translated to accommodate the constructs present in the Java language.

2.2 Related Work

2.2.1 PyCG

PyCG, as described by Salis, et al. (2021), is a static analysis tool used for generating call graphs for Python programs. Its functionalities allow for the identification of the control flow of a program, making it particularly useful for dependency impact analysis (Salis, et al., 2021). To generate a call graph, PyCG builds a graph structure that shows the assignment relationships between identifiers within the program (Salis, et al., 2021). This graph construction is facilitated by analysis of the intermediate representation of the program (Salis, et al., 2021). Using this graph, the tool is then able create its call graphs. In contrast to other static analysers, PyCG can manage Python characteristics such as higher-order functions, modules, function closures and multiple inheritance (Salis, et al., 2021).

2.2.2 SonarQube

SonarQube is popular automated static code analysis tool that assists developers by ensuring software within a codebase is of the highest quality. The tool currently provides support for over 30 programming languages and integrates into continuous integration/continuous development (CI/CD) workflows (SonarQube, n.d.) . SonarQube conducts its analysis using a set of rules which when violated signify that that code is no longer “Clean Code” (SonarQube, n.d.). These rulesets are defined in a built-in or user-defined “Quality profile” that has been set up for the source code language being analysed (SonarQube, n.d.). When the tool is run on code, these rules are checked against to find possible deviations from what is specified in the Quality profile. If a rule violation occurs, it suggests that the code contains a bug, vulnerability, or code smell (SonarQube, n.d.). In the context of SonarQube, a bug refers to some coding defect which may create a runtime error, a vulnerability refers to a weakness in the code which may be exploited by an attacker and code smell refers to an aspect of the code which compromises it maintainability (SonarQube, n.d.). When SonarQube identifies one of these issues, the tool notifies developers through email, the tool’s UI or as part of a pull/merge requests (SonarQube, n.d.). Using this feedback, developers can then rectify the identified problems, ensuring that their code remains reliable, maintainable, and secure (SonarQube, n.d.).

In addition to providing feedback on the quality of code through profiles, SonarQube allows developers to define and configure “Quality Gates” into their CI/CD pipeline (SonarQube, n.d.). Quality gates are a quality guideline which checks that raised issues have been corrected before code is ready for release (SonarQube, n.d.). Therefore, providing further verification that code meets the standards specified by the tool’s users.

2.2.3 Understand by SciTools

Understand by SciTools provides a set of tools designed to statically analyse software, aimed at producing various visualisations and reports in an effort to eliminate bugs and allow for better code comprehension. This multi-tool application provides users with a means to view and visualise dependencies between software components in target software (Understand by SciTools, n.d.). It facilitates the creation of various graphs, such as UML diagrams, control flow diagrams and call trees, for users to better understand the structure and interactions occurring in code (Understand by SciTools, n.d.). Additionally, the application runs automated code checking through its “CodeCheck” feature, which ensures code satisfies user defined or published coding standards such as MISRA C (Understand by SciTools, n.d.). Moreover, it provides version control facilities which allow for analysis of version differences and visualisation of modification history (Understand by SciTools, n.d.). By providing a large range of insights, Understand by SciTools aims to assist developers in comprehending code organisation and its interactions, a characteristic that is particularly useful for legacy code lacking proper documentation (Understand by SciTools, n.d.). Furthermore, the information extracted helps developers better maintain their system and make informed decisions throughout its evolution (Understand by SciTools, n.d.).

3 Requirements Specification

Proceeding the development the project objectives and investigation into inspect4py and related work, the final properties of Inspect4j were captured in the following software requirements:

Functional Requirements

Given the input and output path to a repository, for each file (where applicable):

1. The system must be able to detect and extract basic information about a method. This including:
    i. The name of the method.
    ii. The access and non-access modifiers of the method.
    iii. The parameter names of the method.
    iv. The parameter types of the method.
    v. The return types of the method.
    vi. The line number interval of the method, denoting the starting to end line of method block.
2. The system must be able to detect and extract basic information about a class. This including:
    i. The name of the class.
    ii. The access and non-access modifiers of the class.
    iii. The type parameters of the class.
    iv. The name of the interfaces the class implements.
    v. The parent class from which the class inherits.
    vi. The line number interval of the class, denoting the interval from the starting line to end line of the class definition.
3. The system must be able to extract basic information about an interface. This includes:
    i. The name of an interface.
    ii. The access and non-access modifiers of the interface.
    iii. The type parameters of the interface.
    iv. The name of the interfaces that interface extends.
    v. The line number interval of the interface,
4. The system must be able to detect and extract information about lambda expressions. This includes:
    i. The lambda's parameter and its type
    ii. The lambda's full body
    iii. The lambda's line number interval, denoting the interval from the starting line to end line of the interface definition.
5. For the above entities, the system must be able to detect and extract Javadoc comments, including the descriptions and comment tags.
6. The system must be able label the classes according to whether they are top-level, static nested, inner, or local classes.
7. The system must be able to link nested and inner classes to the top-level classes they belong to.
8. The system must be able to link local classes to the methods they belong to.
9. The system must be able to link nested interfaces to the classes they belong to.
10. The system must be able to link methods to the classes they belong to.

11. The system must be able to execute the entire extraction process both when supplied with a single java file and provided with an entire repository.
12. The system must be able to link lambda expressions to the methods that they appear in.
13. The system must be able to link the method references to the methods or classes that they appear in.
14. The system must be able to link abstract methods to the interfaces they belong to.
15. The system must be able to link nested interfaces and nested classes to interfaces they belong to.
16. The system must be able to extract and store information from the contents of the body of the method, including:
    i. The return statements of the method.
    ii. The direct method calls in the method (i.e. methods that are directly referred to by their name).
    iii. The assignment methods in the method (i.e. called methods whose returned result is stored to a variable).
17. The system must be able to extract and store information from the contents of the body of the method reference, including:
    i. The containing entity of the method reference, such as containing class or containing type, which refers to the part within the brackets preceding the "::".
    ii. The identifier such as the static or non-static method name, which refers to the part within the brackets proceeding the "::".
18. The system must be able to detect and extract file metadata, including:
    i. Whether a file contains a main method and if the method calls another method whose purpose is to be the main starting point of the file's execution
19. The system must be able to detect and extract information about the dependencies or import statements within a file, including:
    i. The name of the imported entity.
    ii. The package or class in which the imported entity resides.
    iii. Whether the imported entity is internal or external to the repository.
    iv. Whether the imported entity is a class, interface, or static member.
20. The system must be able to detect and extract information about the requirements of the repository (managed by Maven), including:
    i. The version of java specified to run the target software within the repository.
    ii. Then name and version of other packages and libraries required to run the target software within the repository.
21. The system must be able to detect whether or not a file is dedicated to testing code within the repository.
22. The system must be able to write the information extracted about each java file to respective Json files.
23. The system must be able to construct and print the directory tree for the repository hierarchy.

4 Software Engineering Process

The project naturally took on an a more plan driven approach to its development. Although a more iterative approach could have allowed for developmental flexibility and adaptability, the objectives of this project are modelled from an existing software system, that already has a clearly defined set of functionalities and features.

In the early stages of development, most of the time was spent expanding on the specific details of the objectives, identifying the structure and architecture the project would use, and researching to learn more about inspect4py and relevant libraries. This then transitioned into the implementation phase where features were developed incrementally, starting with the foundational software components, and then adding functionalities on top of this. This was finally concluded with a testing, verification, and evaluation phase.

5 Ethics

As discussed in the Description, Objectives, Ethics and Resource document submitted for this project, there are no significant ethical considerations that needed to be addressed in the design and development of Inspect4j. Although the framework accesses repositories available on GitHub, Inspect4j does not collect or store any personal or confidential information associated with the repository or its data. Therefore, diminishing any ethical concerns which may arise from its use.

A copy of the self-assessment form is included in the Appendix of this report.

6 Design and Implementation

The purpose of this part of the report is to describe the overall design and implementation of Inspect4j. In addition, it aims to motivate the decisions made, highlighting the choice in structure of the software system.

As mentioned, static code analysis software commonly performs its tasks using the abstract syntax tree representation of source code. Hence, during the initial design stages, an investigation was conducted to pinpoint Java libraries or tools that perform the conversion between source code into an AST. Although the primary aim of the investigation was to select a library that would be used by Inspect4j, it was also an opportunity to understand how the framework's architecture and structure would have to account for the library's integration.

From the results of the investigation, it was decided that the JavaParser library was the most suitable for performing this task. The JavaParser library is an open-source library which, in addition to parsing Java code into an AST, provides various functionalities that allow for exploration and analysis of ASTs (Smith, et al., 2023). Despite other Java libraries providing similar functionalities, JavaParser was adopted because it offers "Visitor Support" (Smith, et al., 2023). Visitor Support aims to allow developers to specify specific entities or aspects of the source code they desire to search for within the tree. The library then provides operations that conduct tree traversal with these aspects in mind. The key advantage of feature is that does not have to program tree traversal from scratch. This not only saves times but also reduces the susceptible to errors (Smith, et al., 2023).

Inspect4j is designed in such a way that it takes advantage of this Visitor Support. From an architectural perspective, the framework contains a distinct class for each captured entity type. This class is then coupled with a collection class that is responsible for storing all entities of that specific type. It is within the respective collection classes that traversal of the AST is conducted. The process involves "walking of the tree", with the intent of finding all occurrence of nodes representing the entity type that collection class aims to gather. Each time a relevant node occurs during the tree search, Inspect4j transforms the node to an alternative object representation and adds it to a list stored within the collection class. Through Visitor Support, the hard task of traversing is handled by JavaParser, and it is up to Inspect4j to deal with expressing which entity types and information it desires to search for.

JavaParser achieves Visitor Support through its use of an Adaptor Pattern (Smith, et al., 2023). The library provides a set of Visitor Adaptor classes which each contain a "visit" method for every node or entity type seen in the Java language. During traversal, when a node is processed the relevant "visit" method is get executed. Applications which use JavaParser can create child class of one of the Visitor Adaptor classes and override the "visit" method for the node types that are of interest (Smith, et al., 2023). By overriding these methods, Inspect4j can specify the entity type it interested in at a given moment and perform operations, such as addition to a class collection list, whenever the specified Java entity or construct appears whilst traversing the tree. This is demonstrated in Listing 4, which provides an example of a user-defined sub class of Visitor Adaptor class taken from the Interface collection class in the Inspect4j project. This subclass contains an overridden visit method for an interface declaration node which specifies that when this node is encountered it should be added to a list. When the node is added to the class collection list, the declaration is not added as is. Instead, Inspect4j extracts specific pieces of information from this node and constructs a new object, which then gets added to the list.

```
private static class InterfaceDeclarationCollector extends VoidVisitorAdapter<List<Interface>> {
    @Override
    public void visit(ClassOrInterfaceDeclaration intDecl, List<Interface> collection) {
        super.visit(intDecl, collection);
        if(intDecl.isInterface()){
            collection.add(new Interface(intDecl));
        }
    }
}
```

Listing 4: Visitor Adaptor subclass containing overridden visit method for an Interface node.

6.1.1 RepoPyOntology translation and Extraction of Software Features

As outlined in Section 2.1.3.1, the entities and relationships captured by Inspect4j should mirror those seen in the RepoPyOnto ontology. Currently, Inspect4j achieves a subset of the functionalities provided by Inspect4py, which implies that some entity and relationship types observed in the ontology are not covered by Inspect4j. Furthermore, some builtin Java constructs differ from those in Python, resulting in replacements and extensions to the ontology having to be made.

With this in mind, Table 3 and Table 4 resent an adaptation of entity and relationship sets seen in the RepoPyOnto ontology. These tables illustrate the entity and relationship sets upon which Inspect4j bases its analysis.

The adaptation introduces new entities such as Interface, library, and constructor. Although a constructor is stated to be a distinct entity, Inspect4j treats it as though it is a method. In terms

of properties, the adaptation introduces new properties such as classes, interfaces, and methods have access and non-access modifiers properties to specify the visibility and scope of these entities. In terms of relationships, the “Parameterises” relationship was added to capture the notion of Generic classes and interfaces. In addition, the “implements” relationship was added to link an interface to classes and implemented interfaces. Docstring has been substituted with JavaDoc and argument has been swapped out for parameter. Inspect4j does not scan through README files to extract any of its information. However, the framework does examine POM files to gather Java version and library related information.

Entity	**Properties**
Repository	name, full_name, description, type....
POM file	Path, content
Library	name, full_name, description, type...
Directory	name, path, parent_path,
Package	name, canonical_name, parent_package, path, parent_path, external, inferred
Class	Name, canonical_name, access_modiers, non_access_modifiers, type, min_line_number, max_line_number,
Interface	name, canonical_name, access_modifiers, non_access_modifiers, category, min_line_number, max_line_number,
Method	Name, type, access_modifiers, non_access_modifiers, builtin, source_code, ast, min_line_number, max_line_number
Constructor	Name, type, access_modifiers, canonical_name, source_code, ast, min_line_number, max_line_number
Variable	Name, canonical_name, type, inferred, access_modifiers, non_access_modifiers
Parameter	Name, type
Return Value	Name, type
Body	Source_code
Javadoc Comment	Description
Javadoc Parameter	Name, description
Javadoc Return Value	Name, description
Javadoc Exception	Type, description

Table 3 Java entity set as captured by Inspect4j.

Relationship	Source	Destination
Requires	Repository	Library
Contains	Repository, Package, Library, Class, Interface, Method, Constructor	Directory, Class, Interface, Method, Package, Variable, Body
Imports	Class	Class, Interface, Variable, Method
HasMethod	Class, Interface	Method
Extends	Class, Interface	Class, Interface
Implements	Class	Interface
HasParameter	Method, Constructor	Variable
Documents	JavaDoc Comment	Method, Class
Describes	JavaDoc Comment	Javadoc Parameter, Javadoc Return Value, Javadoc Exception
Calls	Method, Constructor	Method, Constructor
Parameterises	Class, Interface	Class, Interface

Table 4 Java relationship set as captured by Inspect4j.

The information expressed in the tables provide a concise formal interpretation of the software features extracted by Inspect4j. The software features as extracted and presented in the output JSON files produced by the framework are as follows.

Inspect4j extracts characteristics at the *File* level. For each of java file within a repository, the framework records the following software features:

- *path*: the path of the file.
- *fileNameBase*: name of the file without its file extension.
- *extension*: the file extension.
- *doc:* the Javadoc documentation at the file level. This is often, the Javadoc comment for the public class or interface in the file.

Inspect4j extracts characteristics at the *Dependency* level. For each of file dependency occurring in a file, the framework records the following software features:

- *import*: the name of the class, interface, or static member being imported.
- *type*: states whether the imported entity is internal or external. The entity is internal if it is contained within the repository being analysed. Otherwise, it is external.
- *from_package*: the name of the package containing the imported entity.
- *type_element:* states whether the imported entity is a class, interface, or static member.

Inspect4j extracts characteristics at the *Method* level. For each method occurring in a file, the framework records the following software features:

- *name:* the name of the method.
- *args:* the set of parameters passed into the method. For each parameter, it also extracts the class type of the parameter (as *arg_types*).
- *return_type:* the return type of the method.
- *returns:* the set of all return statements appearing in the method.

- *access_modifiers:* the access modifier applied to the method. This value will either be “private”, “public”, “protected” or “default”.
- *non_access_modifiers:* the set of non-access modifiers applied to the method. Values in the set will either be “final”, “abstract” or “static”. if there are no non access modifiers restricting the scope of the method’s access, then the set will contain a “none” element to signify this. If the method is contained in the interface and no implementation is provided for the method, it is implicitly abstract. Therefore, by default the set will only contain the value “abstract”.
- *doc: the* Javadoc documentation at the method level.
- min_lineno: the line number which contains the start of the method block.
- max_lineno: the line number which contains the end of the method block.
- *calls:* the set of direct method calls made in the method block.
- *lambdas:* the set of lambda expressions in the method block.
- *store_vars_calls:* the set of assignment methods in the method block. An assignment method is a method where when it is called, its return value is stored in a variable.
- *local_classes:* the set of local classes contained within the method block.

Inspect4j extracts characteristics at the level of a *Lambda Expression*. For each lambda expression in a file, the framework stores the following software features:

- *args:* the set of parameters passed into the lambda expression. If the parameter has explicitly been casted to specific type, it also extracts the class type of the parameter (as *arg_types*).
- *returns:* the set of all return statements appearing in the lambda expression.
- *body*: the string representation of the full lambda expression body.
- *min_lineno*: the line number which contains the start of the lambda expression block.
- *max_lineno*: the line number which contains the end of the lambda block.

Inspect4j extracts characteristics at the level of a *Method Reference.* For each method reference appearing in a file, the framework stores the following software features:

- *containing_entity*: the name of the class, object or type used to invoke the method being called.
- *Identifier:* the name of the method being called.

Inspect4j extracts characteristics at the *Class* level. For each class identified, the framework records the following software features:

- *name:* the name of the class.
- *extend:* the superclass the class inherits from.
- *implement:* the set of interfaces the class implements.
- *type_params:* the set of type parameters for the class. The parameters can either be a named non-primitive type or an unnamed type variables (i.e. represented by an upper-case letter).
- *access_modifiers:* the access modifier applied to the class. This value will either be “private”, “public”, “protected” or “default”.
- *non_access_modifiers:* the set of non-access modifiers applied to the class. A value in the set will either be “final”, “abstract” or “static”. if there are no non access modifiers restricting the scope of the class’s access, then the set will contain a “none” element to signify this.

- *doc: the* Javadoc documentation at the class level.
- *min_lineno*: the line number which contains the start of the class block.
- *max_lineno*: the line number which contains the end of the class block.
- *methods:* the set of methods contained in the class.
- *store_vars_calls:* the set of assignment methods *in the class (outside of a method)*.
- *nested_interfaces:* the set of nested interfaces in the class.
- *inner_classes:* the set of inner classes contained within the class. In this context, an inner class is a non-static nested class.
- *static_nested_classes:* the set of static nested classes contained within the class.

Inspect4j extracts characteristics at the *Interface* level. For each identified interface, the framework stores the following software features:

- *name:* the name of the interface.
- *extend:* the set of interfaces the interface extends.
- *type_params:* the set of type parameters for the interface. The parameters can either be a named non-primitive type or an unnamed type variable.
- *access_modifiers:* the access modifier applied to the class. This value will either be "public" or "default".
- *doc: the* Javadoc documentation at the interface level.
- *min_lineno*: the line number which contains the start of the interface block.
- *max_lineno*: the line number which contains the end of the interface block.
- *methods:* the set of methods contained in the interface.
- *nested_interfaces:* the set of nested interfaces in the interface.
- *nested_classes:* the set of nested classes in the interface.

It must be noted that interfaces do not need to contain a set of non-access modifiers because they are always abstract and can never explicitly be declared as static or final. However, their methods may be static.

Inspect4j extracts characteristics at the level of a *Main* method. At this level, the framework aims to identify whether a file contains an entry point. More specifically, the framework records the following software features:

- *main_flag:* states whether there is a main method in the given class.
- *main_method:* provides the first method called within the main method. The first method called is assumed to be the method that actually triggers the execution of the target software or the code in the file in which the main method is contained.

The information above captures the key set used in the JSON files produced by Inspect4j. These keys aim to be similar to those used in Inspect4py in order to maintain consistency between the frameworks. This makes it easier for users who have gained familiarity with one of the frameworks to use the other without having to learn a new set of terminology.

Inspect4j extracts characteristics at the level of a project *Requirement*. This functionality is only available to repositories that are managed by Apache Maven. The justification for this choice is supplied in Section 6.1.3. It is important to mention that the features for a requirement are not written to JSON file in the output directory. Instead, they are written to a requirements.txt file contained in the same directory.

For each required library or package identified, the framework stores the following software features:

- *Java version*: the name of the required java version for compilation of the target software.
- Java target version: the name of the required java version used to run the compiled target software.
- *name:* the name of the required library or package.
- *version*: the version of the required library or package.

Inspect4j extracts characteristics about the *File Hierarchy*. For each file and directory in the repository, the framework builds and stores the organisation of the repository as String. The user is then able to print this string.

As mentioned earlier, each entity type listed will have a corresponding class responsible for creating objects of their type and storing the software features extracted in those objects. For the entity types that may appear more than once in a file, a corresponding collection class will contains a list of those instances.

6.1.2 Capturing Entity Relationships

A by-product of the producing distinct collections or lists for each entity identified by Inspect4j, is that these entities lose awareness of the relationships between each other. In order to restore the relationships captured in the original tree structure, each entity class contains one or more methods designed to locate the parent of the entity and establish a "pointer" to that parent. An illustration of this functionality can be seen in the Method entity shown in Listing 5.

```
private ParentEntity<ClassOrInterfaceDeclaration> findParentClassInterface(CallableDeclaration<?> md) {
    if (md.findAncestor(...types:ClassOrInterfaceDeclaration.class).isPresent()) { // if the method is inside a
                                                                                    // class/interface
        ClassOrInterfaceDeclaration parentIC = md.findAncestor(...types:ClassOrInterfaceDeclaration.class).get();
        if (parentIC == null)
            System.out.println(x:"Could not find parent for this method");
        if (parentIC.isInterface()) { // if the parent is an interface
            return new ParentEntity<ClassOrInterfaceDeclaration>(parentIC, EntityType.INTERFACE);
        } else {
            return new ParentEntity<ClassOrInterfaceDeclaration>(parentIC, EntityType.CLASS);
        }
    }
    return null;
}
```

Listing 5: Method that finds the class or interface that a given method belongs to.

When the method object gets created in the Visitor Adaptor subclass of the method collection class, the constructor of the Method Class will call on the findParentClassInterface method to retrieve the "parent" of the method entity. The ParentEntity returned in the method is assigned to a field within the Method object, creating a pointer to this parent. At a later stage, the parent of the method will run an operation which iterates over the method list in the collection class to find its children. In other words, it checks if its declaration is the same as that stored in the ParentEntity object associated with the child. This is illustrated in Listing 6, wherein the class for the Class entity contains a findMethods method, which iterates over the method collection and checks if its declaration matches the one stored in the given method object. If there is a match, the method is added to a list contained in the parent object signifying that the method belongs to the class.

```java
/**
 * Using MethodCollection, Finds and stores the methods that belong to this class
 * It is responsible for (where appropriate) linking the methods to their parent class
 * @param mds The MethodCollection object
 */
public void findMethods(MethodCollection mds) {
    for (Method md : mds.getMethods()) {
        ParentEntity<?> methodParent = md.getParent();
        if (methodParent != null && methodParent.getEntityType() == EntityType.CLASS) { // if the method is inside a
                                                                                         // class
            if (methodParent.getDeclaration() == declaration) { // if this class is the parent of the method
                methods.add(md);
            }

        }
    }
}
```

Listing 6: Class finds all child method entities.

6.1.3 Requirements Extraction

One of the key project requirements is to be able to compile a list of required libraries. As reported in the fifth annual Developer Ecosystem survey conducted by JetBeans in 2021, Apache Maven is one of the most popular build automation and project management tool with around 72% of developers stating it is their preferred system (JetBeans, 2021). Due to its wide usage, the component which extracts project requirements is designed and implemented with the Maven dependency management system in mind.

All Maven projects must contain at least one Project Object Model (POM) file. This file act as a configuration file for the project and references required internal and external libraries and packages needed to build and run the project. Using the Maven Model API, Inspect4j is able to scan through and retrieve project requirements from all POM.xml files contained in a maven managed repository. This information is then written to a requirements.txt file for Inspect4j users to view.

7 Evaluation and Critical Appraisal

This section of the report provides an evaluation of Inspect4j against the set of objectives proposed in the DOER (as seen in Section 1.1). This section also critically assesses the contributions made in this project against Inspect4py, the framework from which Inspect4j is derived.

7.1 Testing

To assess the accuracy of Inspect4j and verify that it behaves as expected, evaluations were conducted through unit testing and tests on publicly available GitHub repositories.

7.1.1 Unit Tests

The unit tests created for the framework focus on positive test cases, where each test aims to introduce and test a different set of entity and relationship types to ensure that Inspect4J is successfully capturing these in the output JSON files. These tests can be viewed in the JUnit folder of the test directory in the code submission of this project. The unit tests have been distributed over four different files: BasicTests, DepDocTests, NestedTests, and TypeInheritenceTests. The four files aim to test a group of different characteristics that can be seen in Java source code.

Each test within these files runs Inspect4J on an example Java file, producing a corresponding JSON file with the output of the tools analysis. The JSON file is then read into the test file and stored as a JSON object. This JSON object is compared against another JSON object which contains the expected results. As of when these tests were run, Inspect4J successfully passed all of them.

BasicTests

The unit tests in the BasicTests file focus on extraction of basic code features, such as extraction of a single method or multiple methods in a class or interface.

The tests in the BasicTests operate on the 4 following example files:

1. BasicClassWithMain.java, which contains a single class with only a main method.
2. BasicClassWithOneMethod.java, which contains a single class with one method.
3. BasicClassWithMultipleMethods.java, which contains a single class with multiple methods (including a main method).
4. BasicInterfaceWithMethods.java, which contains a single interface with multiple methods.

A few examples from the BasicTests file will be examine. As similar entities and relationships are seen in example files 1-3, the following example will only examine the test for BasicClassWithMultipleMethods. This test aims to check that Inspect4J can successfully detect the presence of multiple methods and report back relevant information. The example java file is seen in Listing 7 and sections of the output JSON file are shown in Listings 8 – 13.

```java
package test_files.test_basic;

import java.util.List;

You, 3 weeks ago | 1 author (You)
public class BasicClassWithMultipleMethods {

    private String catName;
    private String dogName = dogNameSetter();

    Run | Debug
    public static void main(String[] args) {
        System.out.println(x:"Hello World!");
    }

    public BasicClassWithMultipleMethods(String catName) {
        this.catName = catName;
    }

    public String cat(String name, int age, String breed, List<String> list) {
        System.out.println("Cat name:" + name);
        System.out.println("Cat age:" + age);
        System.out.println("Cat bree:" + breed);
        String dog;
        dog = dogNameSetter();      You, 3 weeks ago • stored variable calls in memb
        String dog2 = dogNameSetter();

        return "abdbf";

    }

    public static String bat(String name, int age, String breed) {
        System.out.println("bat name:" + name);
        System.out.println("bat age:" + age);

        if (age > 2) {
            return "yes";
        } else {
            return "no";
        }

    }

    private void mat(String name, int size) {
        System.out.println("mat name:" + name);
        System.out.println("mat size:" + size);
    }

    private String dogNameSetter() {
        return "Sam";
    }
}
```

Listing 7: BasicClassWithMultipleMethods example java file.

Below is a selection of snippets from the output JSON file for BasicClassWithMultipleMethods. The full JSON file is available in the source code submission.

```
{
    "file": {
        "path": "C:\\Users\\OneDrive\\Documents\\4th year\\CS4099 -
Dissertation\\Dissertation\\Inspect4j\\demo\\src\\test\\java\\test_files\\test_bas
ic\\BasicClassWithMultipleMethods.java",
        "fileNameBase": "BasicClassWithMultipleMethods",
        "extension": "java"
    },
```


Listing 8: Snippet of JSON file for BasicClassWithMultipleMethods test which shows basic file information (i.e. path, file name base and extension).


```
    "classes": {
        "BasicClassWithMultipleMethods": {
            "access_modifier": "public",
            "non_access_modifiers": [
                "none"
            ],
            "min_max_lineno": {
                "min_lineno": 5,
                "max_lineno": 50
            },
            "store_vars_calls": {
                "dogName": "dogNameSetter"
            },
             "methods": [
```


Listing 9: Snippet of JSON file for BasicClassWithMultipleMethods test which shows correct detection of class and its metadata (such as class name, access and non-modifiers, and line intervals).

```
            "methods": [
                {
                    "main": {
                        "access_modifier": "public",
                        "non_access_modifiers": [
                            "static"
                        ],
                        "args": [
                            "args"
                        ],
                        "arg_types": {
                            "args": "String[]"
                        },
                        "return_type": "void",
                        "min_max_lineno": {
                            "min_lineno": 10,
                            "max_lineno": 12
                        },
                        "calls": [
                            "System.out.println"
                        ]
                    }
```


Listing 10: Snippet of JSON file for BasicClassWithMultipleMethods test which demonstrates identification of (main) method and its metadata (including name, access and non-access modifiers, parameter name and type, return type and line interval)


```
                {
                    "BasicClassWithMultipleMethods": {
                        "access_modifier": "public",
                        "non_access_modifiers": [
                            "none"
                        ],
                        "args": [
                            "catName"
                        ],
                        "arg_types": {
                            "catName": "String"
                        },
                        "return_type": "void",
                        "min_max_lineno": {
                            "min_lineno": 14,
                            "max_lineno": 16
                        }
                    }
```


Listing 11: Snippet of JSON file for BasicClassWithMultipleMethods which demonstrates identification of (Constructor) method and its metadata (including name, access and non-access modifiers, parameter name and type, return type and line interval)

```
            {
                "cat": {
                    "access_modifier": "public",
                    "non_access_modifiers": [
                        "none"
                    ],
                    "args": [
                        "name",
                        "list",
                        "age",
                        "breed"
                    ],
                    "arg_types": {
                        "name": "String",
                        "list": "List<String>",
                        "age": "int",
                        "breed": "String"
                    },
                    "return_type": "String",
                    "returns": [
                        "abdbf"
                    ],
                    "min_max_lineno": {
                        "min_lineno": 18,
                        "max_lineno": 28
                    },
                    "calls": [
                        "System.out.println",
                        "System.out.println",
                        "System.out.println",
                        "dogNameSetter"
                    ],
                    "store_vars_calls": {
                        "dog": "dogNameSetter",
                        "dog2": "dogNameSetter"
                    }
                }
            },
```


Listing 12: Snippet from the output JSON file of BasicClassWithMultipleMethods test which demonstrates identification of method and its metadata (including name, access and non-access modifiers, parameter name and type, return type and line interval).

```
        }
    },
    "main_info": {
        "main_flag": true,
        "main_method": "System.out.println"
    }
}
```


Listing 13: A Snippet from the output JSON file for BasicClassWithMultipleMethods test demonstrates correct detection of presence of a main method and the main entry/starting method call.

As demonstrated by the snippets from the JSON file above, Inspect4j accomplishes the basic objectives of identifying and extracting basic metadata about methods and classes. Moreover, the framework can successfully identify the presence of main method and report this as part of the file metadata. Similarly, Inspect4j can conduct basic analysis of an interface, as demonstrated in the unit test for the BasicInterfaceWithMethods file.

This file contains an interface with 2 abstract methods (as shown in Listing 14) and a section of the output JSON file is in Listing 15.

```
package test_files.test_basic;        You, 3 weeks ago • update

import java.util.List;

You, 2 months ago | 1 author (You)
public interface BasicInterfaceWithMethods {
    String bat(String name, int age, String breed, List<String> list);
    String printAddress(String address);
}
```

Listing 14: BasicInterfaceWithMethods example java file.

```
"interfaces": {
    "BasicInterfaceWithMethods": {
        "access_modifier": "public",
        "methods": [
            {
                "bat": {
                    "access_modifier": "default",
                    "non_access_modifiers": [
                        "abstract"
                    ],
                    "args": [
                        "name",
                        "list",
                        "age",
                        "breed"
                    ],
                    "arg_types": {
                        "name": "String",
                        "list": "List<String>",
                        "age": "int",
                        "breed": "String"
                    },
                    "return_type": "String",
                    "min_max_lineno": {
                        "min_lineno": 6,
                        "max_lineno": 6
                    }
                }
            },
            {
                "printAddress": {
                    "access_modifier": "default",
                    "non_access_modifiers": [
                        "abstract"
                    ],
```


Listing 15: A Snippet from the output JSON file for BasicInterfaceWithMethods test which shows identification of an interface, its methods, and their respective metadata.

DepDocTests

The unit tests in the DepDocTests file focus on extraction of documentation and dependencies.

The tests in the DepDocTests operate on the 4 following example java files:

1. BasicClassWithDependencies.java, which contains a single class with external dependencies only. The import statements reference specific classes, interfaces, or static members of those dependencies.
2. BasicClassWithInternalDependencies.java, which contains a single class with internal dependencies.
3. BasicClassWithDependenciesAsterisk.java, which contains a single class with external dependencies. The import statements point to entire packages rather than a specific classes, interfaces, or static members.
4. BasicClassWithJavaDoc.java, which contains a single class with Javadoc comments.

A few examples from DepDocTests file will be examine. The following example will examine the test for BasicClassWithInternalDependencies. This test aims to check that Inspect4J can successfully detect the presence of internal dependencies and report back relevant information. The example java file is seen in Listing 7 and sections of the output JSON file are shown in Listings 16 and 17.

```java
package test_files.test_doc_and_dependencies;

import test_files.test_inheritence_type_params_and_superclasses.*;
import test_files.test_nested_classes_interfaces.BasicClassWithNestedClasses;

You, 2 weeks ago | 1 author (You)
public class BasicClassWithInternalDependencies {
    private String name;
}
```

Listing 16: BasicClassWithInternalDependencies example Java file.


```json
"dependencies": [
    {
        "from_package": "test_files.test_inheritence_type_params_and_superclasses",
        "import": "GenericClass",
        "type": "internal",
        "type_element": "class"
    },
    {
        "from_package": "test_files.test_inheritence_type_params_and_superclasses",
        "import": "GenericClassParent",
        "type": "internal",
        "type_element": "class"
    },
    {
        "from_package": "test_files.test_inheritence_type_params_and_superclasses",
        "import": "GenericTypeInterface",
        "type": "internal",
        "type_element": "interface"
    },
    {
        "from_package": "test_files.test_inheritence_type_params_and_superclasses",
        "import": "GenericTypeInterfaceExtendsInterface",
        "type": "internal",
        "type_element": "interface"
    },
    {
        "from_package": "test_files.test_nested_classes_interfaces",
        "import": "BasicClassWithNestedClasses",
        "type": "internal",
        "type_element": "class"
    }
],
```


Listing 17: A snippet from the output JSON file produced from the BasicClassWithInternaDependecies test, which demonstrates that the tool can successfully detect when dependencies are internal and, in the case where a package is referenced rather than a specific entity, it reports back all entities in that package.

Similarly, Inspect4j can extract Javadoc comments for a class interface as demonstrated in the unit test for the BasicClassWithJavaDoc file. A section of the example java file and output JSON file is visible in Listing 18 and Listing 19.

```java
/**
 * Prints cat details
 * @param name - (int) cat name
 * @param age - (int) cat age
 * @param breed - (String) cat breed
 * @param list
 * @return
 */
public String cat(String name, int age, String breed){
    System.out.println("Cat name:"+name);
    System.out.println("Cat age:"+age);
    System.out.println("Cat bree:"+breed);
    return "printed cat details";

}
```

Listing 18: A snippet from the BasicClassWithJavaDoc example java file

```
    "cat": {
                "doc": "Prints cat details. [tags = [@PARAM param - (int) cat
name, @PARAM param - (int) cat age, @PARAM param - (String) cat breed, @PARAM
param , @RETURN return  ]",
                    "access_modifier": "public",
                    "non_access_modifiers": [
                        "none"
                     ],
                     "args": [
                         "name",
                         "age",
                         "breed"
```

Listing 19: A snippet from the output JSON file for the method shown in Listing 18. This listing demonstrates that Inspect4J can successfully identify and report JavaDoc comments.

TypeInheritenceTests

The unit tests in the TypeInheritenceTests file focus on testing that Inspect4J can extract information about inheritance relationships, type parameterisation and interface implementation.

The tests in the TypeInheritenceTests operate on these 4 following example files:

1. GenericClassParent.java, which contains a single generic class.
2. GenericTypeInterface.java, which contains a single generic interface.
3. GenericClass.java, which contains a single generic class which extends GenericClassParent.java and implements GenericTypeInterface.
4. GenericTypeInterfaceExtendsInterface.java, which contains a single interface which extends GenericTypeInterface.

For the sake of brevity, the snippets for these tests have not been included. However, the input and output files are available in the test_inheritence_type_params_and_superclasses directory

of the test_files directory of the code submission. The output JSON files seen in this directory show inspect4J can successfully detect and report these features.

NestedTests

The unit tests in the NestedTests file focus on testing that Inspect4J can successfully identify and correctly report that a file contains multiple classes and interfaces. More specifically the tests check if it can correctly detect local, inner, and nested classes and interfaces.

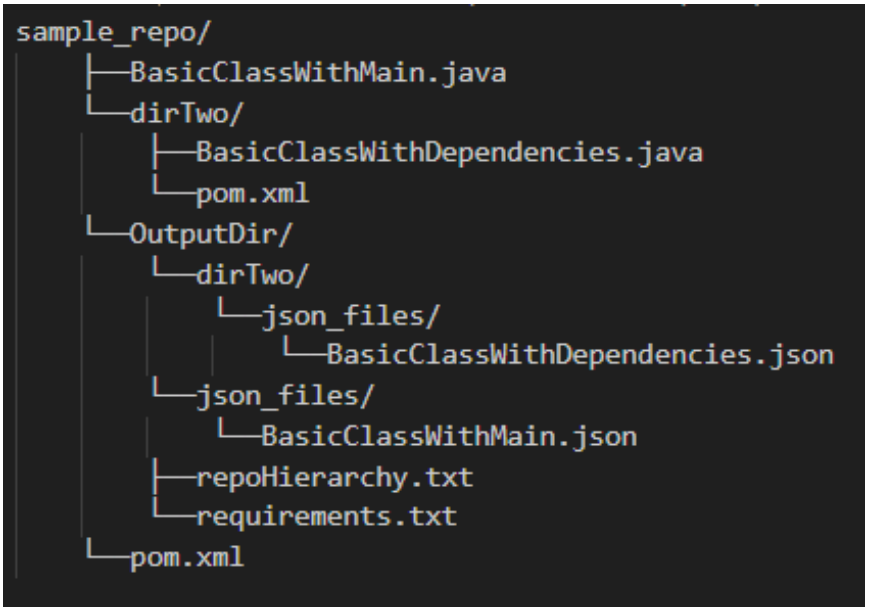
The tests in the NestedTests operate on these 3 following example files:

1. BasicClassWithLocalDefaultClass.java, which contains a public and default class, where the public class contains a method that has a local class.
2. BasicClassWithNestedClass.java, which contains a single top-level class that has an inner and nested class.
3. BasicInterfaceNestedInClass.java, which contains a single top-level class that has a nested interface and an inner class.

For the sake of brevity, the snippets for these tests have not been included. However, the input and output files are available in the test_nested_classes_interfaces directory of the test_files directory in the code submission. The output JSON files seen in this directory show that inspect4J can successfully detect and report these features.

Testing file organisation and requirements extraction

To ascertain the frameworks' ability to accurately document the file organisation within a repository, an example repository with test files was created, and the framework was subsequently executed on it. Furthermore, the example repository contains POM files which are used to evaluate the framework's ability to extract the repository's required libraries and packages. The outcomes of these evaluations are shown in Listing 20 and Listing 21 , which demonstrates Inspect4j's ability to extract requirements and accurately represent the repository hierarchy.



Listing 20: Repository hierarchy for the example repository.

```
Java Version: 17
Java Target Version: 17

com.graphql-java:graphql-java==21.4
org.slf4j:slf4j-api==2.0.12
junit:junit==4.11
com.github.javaparser:javaparser-symbol-solver-core==3.25.5
commons-io:commons-io==2.8.0
info.picocli:picocli==4.7.5
com.github.javaparser:javaparser-core-serialization==3.25.5
com.google.code.gson:gson==2.10.1
org.json:json==20230618
org.apache.maven:maven-model==3.6.3
```

Listing 21: The contents of requirements.txt file for the example repository.

7.2 Testing on GitHub Repositories

The primary objective of Inspect4J is to supply insights into repositories encountered "in the wild". Consequently, conducting tests on 'real' repositories becomes imperative when assessing its performance. By performing real-world testing, it allows for a more diverse set of test scenarios, which many not be addressed by the unit tests. So, Inspect4J was executed on five publicly available GitHub repositories, the links to which are included in the Appendix of this report. The output files resulting from this execution is available in the project's code submission.

Upon reviewing the output files, it is clear that Inspect4J successfully analyses all 5 repositories. However, these tests highlight a few limitations in its implementation. Primarily, the framework fails to account for Enums, thus incorrectly disregarding these files during analysis. Additionally, it does not handle anonymous classes correctly. It is important that these limitations are corrected in the future as these a commonly used constructs which should be capture accurately.

7.3 Requirements

Primary Objectives

The focus of the primary objectives was to gain familiarity with Inspect4py and abstract syntax trees as a data structure. Furthermore, build a minimum viable product (MVP) for analysing class, interface, and method details along with their respective documentation.

The work produced achieves all the primary objectives. This is corroborated by the unit and GitHub repository tests conducted and documented in this report. As described in Part 6, Inspect4j relies heavily on the Visitor Support provided by JavaParser which means that it was crucial to "reconcile" the relationships between entities and correctly capture these relationships in the output JSON files. As part of future work, it would be beneficial to provide users with more specific information about Lambda expressions.

Secondary Objectives

The secondary objectives expand on the MVP by adding the ability to extract file metadata, which includes detecting the existence of a main method, identify file organisation in a repository, and identify dependencies and project requirements.

The current implementation of Inspect4j achieves these objectives, however there is room for improvement. Currently, requirement extraction limited to Apache Maven- managed projects. Ideally, a more agnostic approach could have been taken and should be compatible with code bases that use a variety of project management and build automation tools.

As it stands, Inspect4j can provide the features described in the secondary objectives and this is confirmed by the tests run to validate these features.

Tertiary Objective

The tertiary objective of this project aims to add a feature which allows for recognition of test files. The expectation with tertiary objectives is that these are achieved if all other aspects were achieved without much challenge. Due to time constraints and difficulties whilst attempting to fulfil the primary and secondary objectives, the current version of Inspect4j does not provide support for test file recognition.

7.3.1 Evaluation of the RepoPyOnto ontology translation

The entity and relationship set seen in Table 3 and Table 4 provides a suitable adaptation of the RepoPyOnto ontology and outlines a good range of constructs captured by Inspect4j. The adaptation encompasses many of the shared constructs seen in Python and Java. However, it also introduces new entities, relationships, and properties. As mentioned previously, Inspect4j provides a subset of the feature provided by Inspect4py and so the entities recognised by the framework are limited to those presented in Table 3. A commonly used entity type that is not included in the Table 3 is the Enum data type. One may argue that an Enum is a specialisation of a Class and therefore, may be treated as such. Nonetheless, Inspect4j does not account for Enums.

8 Conclusions

Informed by the work presented by Filgueira & Garijo (2022), the objective of this project was to create a static code analysis which provides comprehensive analysis of Java code repositories. This framework aims to extend the capabilities of Inspect4py to the Java domain, with the goal of supporting individuals during the process of software discovery, adoption, and reuse. The application produced successfully achieves the primary and secondary objectives of the project, allowing users to execute the tool and receive basic analysis of a Java repository.

8.1 Challenges

Due to various challenges faced during the development of this tool, the work presented did not fulfil all the requirements proposed in the DOER. Although some of this can be attributed to time constraints, other factors had played a role in the inability to address these requirements. One of the key challenges faced during the development of Inspect4J was the lack of documentation and support available for JavaParser. A significant proportion of time was spent familiarising oneself with JavaParser and attempting to infer some of its functionalities through trial and error. Therefore, creating delays during the implementation of a few of the software components in this project.

Writing the gathered information to a JSON file proved to be a more challenging task than anticipated. Enforcing consistency in format and terminology with inspect4py was crucial for maintaining coherence. However, enforcing this was challenging because the library which serialises the Java objects created by Inspect4py require a custom serialisers to be written for each entity type. As evidenced by the length of the OutputWriter class in the code submission, it the code for the custom serialisers is neither concise nor elegant. Despite efforts to find an alternative serialisation method, this was unsuccessful. Therefore, consuming a significant time which could have been allocated to more significant project requirements.

Performing comprehensive testing is key to producing high quality and reliable software. Unfortunately, the number of unit tests created for the framework is quite limited. To verify that the information being written to a JSON file is what is expected, a JSON object with the expected data needs to be created. To check the similarity of this JSON object against the JSON file, it was necessary that the JSON object followed the exact structure and formatting of the JSON file. In other words, each test would require one to reconstruct the entire JSON file but in the form of a JSON object. Programming this reconstruction for each test requires a significant amount of time. Therefore, it was decided that for the sake of efficiency fewer tests would be created but these tests would aim to test multiple functionalities at the same time. To compensate for this, the unit tests are supplemented with the tests conducted on the publicly available GitHub repositories.

8.2 Limitations and Future Work

While Inspect4J achieves the primary and secondary objectives presented in Section 1.1, there are areas in which improvements can be made and opportunities for future work. As it stands, the framework does not support Enums neither does it correctly capture anonymous classes. These are commonly used constructs in Java which means that properly capturing them is important. Therefore, in future, one should rectify this and provide proper support for these Java concepts.

In future, it would also be highly advantageous to users to provide support for test file recognition. This does not appear to be too complex a feature to implement. Drawing from the implementation of Inspect4py, test file recognition would likely involve scanning files for test-related annotations (e.g. @Test) and checking for “assert” statements, dependencies that originate from popular testing libraries and use of naming conventions which imply some relation to testing. Offering this functionality to users will further enrich the insights provided by Inspect4J and ensure that consistency is maintained across the two frameworks.

Similarly, providing output which is consistent with what is produced by Inspect4py is important. Currently, Inspect4J produces a single output JSON file for each Java file in the given repository. However, one feature of Inspect4py is that not only does it create individual JSON files, but it also produces a summary JSON file which contains all information found in the separate JSON file. In addition to this, it also provides an HTML version of the JSON files to make easier for users to read and interpret the data extracted. Therefore, in the future, it would also be useful to present the output data in these ways.

Finally, as previously noted, Inspect4J only offers a subset of the functionalities provided by Inspect4py, which means that it is currently limited in the insights it provides users with. Therefore, implementing the remaining functionalities and feature would be highly valuable as it would ensure that users are provided with a diverse range of insights, leading to an even stronger understanding of the Java source code. This enhancement will enhance the effective of Inspect4j for users.

9 References

Ayewah, N. et al., 2008. Using Static Analysis to Find Bugs. *IEEE Software,* 25(5), pp. 22-29.

Chaudhary, A., 2022. *What is Static Code Analysis? Its Limitations and Benefits.* [Online]
Available at: https://www.hatica.io/blog/static-code-analysis/
[Accessed 19 March 2024].

Cornell University Department of Computer Science, 2022. *Building ASTs and Handling Errors.* [Online]
Available at: https://www.cs.cornell.edu/courses/cs4120/2022sp/notes.html?id=ast
[Accessed 29 February 2024].

Elkhalifa, I. & Ilyas, B., 2016. *Static Code Analysis: A Systematic Literature Review and an Industrial Survey,* Karlskrona: Blekinge Institute of Technology.

ESLint, n.d. *Core Concepts.* [Online]
Available at: https://eslint.org/docs/latest/use/core-concepts
[Accessed 11 March 2024].

Filgueira, R. & Garijo, D., 2022. Inspect4py: A Knowledge Extraction Framework for Python Code. *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR),* pp. 232-236.

Filgueira, R. & Williams, C., 2023. RepoGraph: A Novel Semantic Code Exploration Tool for Python Repositories Based on Knowledge Graphs and Deep Learning. *2023 IEEE 19th International Conference on e-Science (e-Science),* pp. 1-10.

Finamore, A., 2021. *Abstract Syntax Trees in Python.* [Online]
Available at: https://pybit.es/articles/ast-intro/
[Accessed 29 February 2024].

JetBeans, 2021. *Java.* [Online]
Available at: https://www.jetbrains.com/lp/devecosystem-2021/java/
[Accessed 19 March 2024].

MathWorks, n.d. *What Is Static Code Analysis?.* [Online]
Available at: https://uk.mathworks.com/discovery/static-code-analysis.html
[Accessed 8 March 2024].

Novak, J., Krajnc, A. & Žontar, R., 2010. Taxonomy of Static Code Analysis Tools. *The 33rd International Convention MIPRO,* Issue
https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=5533417, pp. 418-422.

Racordon, D., 2021. *From ASTs to Machine Code with LLVM.* Cambridge, United Kingdom, Association for Computing Machinery.

Salis, V. et al., 2021. *PyCG: Practical Call Graph Generation in Python.* Madrid, IEEE.

Schiewe, M., Curtis, J., Bushong, V. & Cerny, T., 2022. Advancing Static Code Analysis With Language-Agnostic Component Identification. *IEEE Access,* Volume 10, pp. 30743-30761.

Smith, N., van Bruggen, D. & Tomassetti, F., 2023. *JavaParser: Visited.* s.l.:s.n.

SonarQube, n.d. *Adding coding rules*. [Online]
Available at: https://docs.sonarsource.com/sonarqube/latest/extension-guide/adding-coding-rules/
[Accessed 11 March 2024].

SonarQube, n.d. *Code analysis based on Clean Code*. [Online]
Available at: https://docs.sonarsource.com/sonarqube/latest/user-guide/clean-code/code-analysis/
[Accessed 18 March 2024].

SonarQube, n.d. *Issues*. [Online]
Available at: https://docs.sonarsource.com/sonarqube/9.9/user-guide/issues/
[Accessed 18 March 2024].

SonarQube, n.d. *Quality gates*. [Online]
Available at: https://docs.sonarsource.com/sonarqube/latest/user-guide/quality-gates/
[Accessed 18 March 2024].

SonarQube, n.d. *Quality profiles*. [Online]
Available at: https://docs.sonarsource.com/sonarqube/latest/instance-administration/quality-profiles/
[Accessed 18 March 2024].

SonarQube, n.d. *Security-related rules*. [Online]
Available at: https://docs.sonarsource.com/sonarqube/9.9/user-guide/rules/security-related-rules/
[Accessed 18 March 2024].

SonarQube, n.d. *SonarQube 10.4 Documentation*. [Online]
Available at: https://docs.sonarsource.com/sonarqube/latest/
[Accessed 18 March 2024].

Stratego/XT, n.d. *Chapter 5. Syntax Definition and Parsing*. [Online]
Available at: http://releases.strategoxt.org/strategoxt-manual/unstable/manual/chunk-chapter/tutorial-parsing.html
[Accessed 30 February 2024].

The Apache Software Foundation, n.d. *Maven Model 3.8.2 API*. [Online]
Available at: https://maven.apache.org/ref/3.8.2/maven-model/apidocs/index.html
[Accessed 18 March 2024].

Understand by SciTools, n.d. *Automatically Check Your Code Using Understand*. [Online]
Available at: https://scitools.com/codecheck
[Accessed 18 March 2024].

Understand by SciTools, n.d. *Bring Your Code to Life Using Graphs*. [Online]
Available at: https://scitools.com/graphs
[Accessed 18 March 2024].

Understand by SciTools, n.d. *Understand Features*. [Online]
Available at: https://scitools.com/features
[Accessed 18 March 2024].

Understand by SciTools, n.d. *Understand: Legacy Code Analysis Made Easy.* [Online]
Available at: https://scitools.com/legacy-code
[Accessed 18 March 2024].

Wilkinson, M. D. et al., 2016. The FAIR Guiding Principles for scientific data management and stewardship. *Scientific Data,* 3(1), p. 160018.

10 Appendix

10.1 Ethics Form

UNIVERSITY OF ST ANDREWS
TEACHING AND RESEARCH ETHICS COMMITTEE (UTREC)
SCHOOL OF COMPUTER SCIENCE
PRELIMINARY ETHICS SELF-ASSESSMENT FORM

This Preliminary Ethics Self-Assessment Form is to be conducted by the researcher, and completed in conjunction with the Guidelines for Ethical Research Practice. All staff and students of the School of Computer Science must complete it prior to commencing research.

This Form will act as a formal record of your ethical considerations.
Tick one box

- ☐ **Staff Project**
- ☐ **Postgraduate Project**
- ☒ **Undergraduate Project**

Title of project

Inspect4J – Comprehensive Static Code Analysis for Java Software Repositories

Name of researcher(s)

Nandi Mbimbi Ncube

Name of supervisor (for student research)

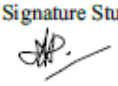
Dr Rosa Filgueira

OVERALL ASSESSMENT (to be signed after questions, overleaf, have been completed)

Self audit has been conducted **YES** ☒ **NO** ☐

There are no ethical issues raised by this project

Signature Student or Researcher



Print Name

Nandi Mbimbi Ncube

Date

11/09/2023

Signature Lead Researcher or Supervisor

Print Name

Rosa Filgueira

Date

11/09/2023

This form must be date stamped and held in the files of the Lead Researcher or Supervisor. If fieldwork is required, a copy must also be lodged with appropriate Risk Assessment forms. The School Ethics Committee will be responsible for monitoring assessments.

Computer Science Preliminary Ethics Self-Assessment Form

Research with secondary datasets

Please check UTREC guidance on secondary datasets (https://www.st-andrews.ac.uk/research/integrity-ethics/humans/ethical-guidance/secondary-data/ and https://www.st-andrews.ac.uk/research/integrity-ethics/humans/ethical-guidance/confidentiality-data-protection/). Based on the guidance, does your project need ethics approval?

YES ☐ NO ☒

** If your research involves secondary datasets, please list them with links in DOER.*

Research with human subjects

Does your research involve collecting personal data on human subjects?

YES ☐ NO ☒

If YES, full ethics review required

Does your research involve human subjects or have potential adverse consequences for human welfare and wellbeing?

YES ☐ NO ☒

If YES, full ethics review required
For example:
Will you be surveying, observing or interviewing human subjects?
Does your research have the potential to have a significant negative effect on people in the study area?

Potential physical or psychological harm, discomfort or stress

Are there any foreseeable risks to the researcher, or to any participants in this research?

YES ☐ NO ☒

If YES, full ethics review required
For example:
Is there any potential that there could be physical harm for anyone involved in the research?
Is there any potential for psychological harm, discomfort or stress for anyone involved in the research?

Conflicts of interest

Do any conflicts of interest arise?

YES ☐ NO ☒

If YES, full ethics review required
For example:
Might research objectivity be compromised by sponsorship?
Might any issues of intellectual property or roles in research be raised?

Funding

Is your research funded externally?

YES ☐ NO ☒

If YES, does the funder appear on the 'currently automatically approved' list on the UTREC website?

YES ☐ NO ☐

If NO, you will need to submit a Funding Approval Application as per instructions on the UTREC website.

Research with animals

Does your research involve the use of living animals?

YES ☐ NO ☒

If YES, your proposal must be referred to the University's Animal Welfare and Ethics Committee (AWEC)

University Teaching and Research Ethics Committee (UTREC) pages
http://www.st-andrews.ac.uk/utrec/

10.2 User Manual

Inspect4j has been tested on Unix and Windows 11(22621.3007) and has been configured to run on Java SE 17.0.

Requirements

To run Inspect4j, this will require JDK17. The project is managed by Maven (3.9.6); however, Maven should not be required to run the program.

Dependencies

Inspect4j uses the following list of libraries/dependencies:

junit:junit==4.11
com.github.javaparser:javaparser-symbol-solver-core==3.25.5
commons-io:commons-io==2.8.0
info.picocli:picocli==4.7.5
com.github.javaparser:javaparser-core-serialization==3.25.5
com.google.code.gson:gson==2.10.1
org.json:json==20230618
org.apache.maven:maven-model==3.6.3

Installation

To install Inspect4j from studres:

1. Download the project's zip file.
2. Unzip the project in one's desire location.

To install Inspect4j from GitHub:

1. Create an Inspect4j directory within one's desired location.
2. Within the Inspect4j directory,
   Run "git clone https://github.com/nandincube/Inspect4j.git"

Execution

To execute Inspect4j on a given repository or file, follow the following instructions:

1. Cd into the demo directory within the Inspect4j repository
2. Run "java -jar target/Inspect4j-1.0-jar-with-dependencies.jar <FILE.java | DIRECTORY> [OUTPUT_DIRECTORY]" to perform analysis on the repository.

   Here, "FILE.java|DIRECTORY" argument refers to the path to the file/repository that one wants to analyse and the "OUTPUT_DIRECTORY" argument refers to the path to the directory where one would like the outputted JSON files to be stored.

   "OUTPUT_DIRECTORY" is an optional argument. If the argument is not provided, Inspect4j will create a new directory which by default is called "OutputDir". This directory will appear in the demo directory or within the directory that the "java -jar target/Inspect4j-1.0-jar-with-dependencies.jar <FILE.java | DIRECTORY> [OUTPUT_DIRECTORY]" command is run.

3. After running the command in step 2, Inspect4j should analyse the given repository and produce a corresponding JSON file for each java file in the repository. These JSON files should be contained in the output directory specified.

   If the repository is a maven project, the output directory should also contain a single requirements.txt file which lists all repository requirements based on the contents of the POM.xml file(s) contained in the repository.
4. Alternatively, if one wants to generate the repository hierarchy tree for a given repository: Within the demo directory, run “java -jar target/Inspect4j-1.0-jar-with-dependencies.jar <DIRECTORY> --tree” which should print the directory tree of the repository to the terminal.

10.3 Testing on GitHub Repositories

The GitHub repositories used for testing are available at the following locations:

1. https://github.com/martinmimigames/little-music-player.git
2. https://github.com/AmbalviUsman/Billing-System.
3. https://github.com/dev-aniketj/WeatherApp-Android.git
4. https://github.com/HouariZegai/Calculator.git
5. https://github.com/PushpinderSinghGrewal/lan-chat-app.git

To view the output files for each repository, these are available in the code submission for this project.