Ministry of Higher Education and Scientific
Research of the People's Democratic Republic of
Algeria

University of Science and Technology Houari Boumediene

Faculty of Computer Science

Multidisciplinary Project

Computer Engineering – Second Year

Research Theme: Empirical Studies of Object-Oriented Software Metrics

Supervised by :

Dr MEKAHLIA Fatma Zohra LAKRID

Jury Members:

Dr. MOULAI

Dr. MEKAHLIA Fatma Zohra

Group: 36

Presented by :

TAIEB BENABBES Nesrine

BOUDIAF Aicha

KAMIRI Lilia

YAHIAOUI Chahinez

BENSLIMANE Salima Lamia

Academic Year: 2024-2025

Acknowledgments

We are deeply thankful to Allah for His guidance and patience, which have been a constant source of strength throughout our academic journey.

Our sincere gratitude goes to all our professors, whose dedication and passion for teaching have inspired us every step of the way. We are also profoundly grateful to our parents for their unconditional love and unwavering support, which have made all of this possible.

A special and heartfelt thank you goes to Ms. Mikahlia, our mentor and supervisor. Her trust in us and her patient guidance have been invaluable. She not only opened doors to new ideas and learning opportunities but also encouraged us to challenge ourselves and think beyond what we thought was possible. Her support has been fundamental to our growth, and we deeply appreciate the guidance she provided throughout this process.

Finally, we would like to warmly thank the jury members for their time, thoughtful evaluation, and constructive feedback, all of which played a crucial role in the successful completion of our project.

Table of Contents

Introduction

In today's rapidly evolving digital world, software development is not just about building applications that work it's about creating clean, efficient, and sustainable code. With increasing system complexity, ensuring the quality of code has become an essential task for developers, researchers, and organizations alike.

Software metrics, particularly object-oriented metrics, play a key role in this quality assurance process. These metrics help identify structural weaknesses, measure the complexity of components, and predict the likelihood of defects. They provide insight into how maintainable, understandable, and reusable the code is making them critical in evaluating and improving software design. It is within this context that our project was born.

We have developed a Java desktop application using Swing, designed to analyze classes from the Defects4J dataset using a carefully selected set of object-oriented metrics. These include values such as coupling between objects, depth of inheritance, number of attributes, method parameters, object instantiations, and more. The results are processed and exported into a structured CSV file, making them easy to read, analyze, and reuse.

The application also features an elegant and user-friendly graphical interface, blending professionalism with interactivity, designed to offer both clarity and comfort for the user. Built using the MVC (Model-View-Controller) architecture, our system is modular, maintainable, and easy to extend.

In the sections that follow, we will explore the tools that made this possible, the logic and implementation strategies, the problems we encountered and solved, and the design choices behind the application from code analysis to user interaction. Welcome to a world where code speaks and metrics reveal its true structure.

I. Foundations of Code Analysis and Object-Oriented Metrics

Analyzing code isn't just a technical necessity it's a doorway to understanding how a software system behaves, evolves, and performs under real-world conditions. Just as a doctor reads health data to diagnose a patient, developers and researchers use metrics to evaluate the health of a software system.

Object-Oriented Programming (OOP) introduces concepts like classes, inheritance, encapsulation, and polymorphism. While powerful, these concepts can lead to complex interdependencies and hidden issues if not handled with care.

That's where object-oriented metrics come in they act as measurable indicators that reflect the internal quality of the code. For example:

- Coupling Between Objects (CBO): Measures how interdependent classes are.
- Depth of Inheritance Tree (DIT): Reflects how deep a class is in the inheritance hierarchy.

These metrics help developers answer crucial questions:

- Is my class too complex?
- Is my code too tightly coupled?
- Can it be maintained or reused easily?

In our application, we not only compute these metrics but also visualize them in a practical way that supports research and development. This bridges the gap between raw code and actionable insight.

Furthermore, by integrating tools like JavaParser and navigating the Abstract Syntax Tree (AST) of Java classes, we are able to deeply understand the structure of code and extract relevant data automatically.

Combined with Defects4J a dataset of real-world buggy and fixed Java classes we offer a practical way to observe how code quality metrics correlate with actual software defects. In summary, our project lies at the intersection of research and real-world application, using theory-driven metrics to support better development practices.

It reflects a broader goal: empowering developers, testers, and researchers to build better software through smart, automated analysis

II. Tools and Technologies

This section presents the tools, libraries, and platforms utilized during the development of our Swing-based software metrics analyzer.

Each component played a crucial role in either the applications implementation, empirical analysis, or team collaboration

II.1 Development and Analysis Tools

Defects4J: A curated collection of real-world Java bugs extracted from open-source projects. It provides both buggy and fixed versions of code, making it an ideal dataset for evaluating how object-oriented metrics correlate with actual software defects.

Maven: A powerful build automation and dependency management tool. It ensured consistent builds, streamlined library integration (e.g., JavaParser), and simplified project structure and lifecycle management.

Strawberry Perl: Required to execute Defects4J scripts on Windows systems. It enabled smooth automation for test execution, bug reproduction, and project setup.

Eclipse IDE: The primary integrated development environment used for coding, debugging, and testing. Its robust Java support allowed seamless integration with tools like Maven and Swing.

Swing Integration:The GUI was fully developed using Swing, providing a simple and effective interface.

JavaParser: A library that parses Java source code into an Abstract Syntax Tree (AST), enabling accurate extraction of static metrics such as class structure, method parameters, and attribute counts.

II.2 Collaboration, Planning, and Problem-Solving Tools

Ubuntu (for Defects4J execution): Ubuntu’s terminal environment was used to install and manage Defects4J, automate test runs, and extract project versions efficiently especially useful in cases where Windows compatibility was limited.

GitHub: Used for version control and issue tracking. It ensured smooth code sharing and progress synchronization between all team members.

Google Meet: Hosted regular team meetings for planning, task delegation, and progress reviews. These virtual sessions played a key role in maintaining group cohesion and alignment throughout the project.

Discord: Served as a real-time communication platform for discussions, announcements, quick updates, and live coding support

Telegram: Used for transferring large files thanks to its generous file size support.

III Our application's Metrics

III.1 JavalyzerX

JavalyzerX (JAX) is a static analysis tool designed to examine the structure of a Java project without executing it. It helps in:

- Assessing code quality.
- Identifying complexity hotspots.
- Tracking project evolution.
- Supporting maintenance, documentation, or optimization.

Metrics collected by JAX and their purpose:

III.1.1 Total Non-Empty Lines (JAX):

- **Definition:**

JAX (Java Analyzer Lines): refers to the number of non-empty lines in a Java file. It includes both lines of code and comment lines (lines that include comments (`//`, `/* ... */`, or `/** ... */`) in a Java file., as long as they are not empty.

- **Why JAX ?:**

It is helpful as it provides a reliable indicator of actual written code, excluding empty or formatting lines, and serves as a baseline for comparing file sizes.

- **Calculation method:**
  - F is a Java source file.
  - L is the set of all lines in F.
  - LV is the set of lines that are empty (i.e., only contain whitespace or tabs).

- **Formula**

$NonEmptyLines\ (F) = |L| - |LV|$.

❖ Comment Lines

- **Definition:**

Counts all lines that are part of comment blocks, including single-line (//), block (/* ... */), and Javadoc (/** ... */) comments.

- **Why it matters:**

Comment density is a strong indicator of code maintainability and documentation quality.

- **Computation:**

* Extract all comment blocks

* Sum the number of lines spanned by each comment

- **Formula:**

CommentLines = Σ|(endLine - startLine + 1)| for each comment block.

III.1.2 Total Number of Class Declarations

- **Definition**:

Counts all non-interface declarations in the file.

- **Why it matters:**

It is helpful as it reveals the number of concrete class definitions, which indicates the structural complexity and modularization of the code.

- **Calculation method:**

  Let F be a Java file.

  Let C be the set of class declarations in F, excluding interfaces.

- **Formula:**

DeclaredClasses(F)=|{c∈F|c is a ClassOrInterfaceDeclaration∧¬c.isInterface()}|

III.1.3 Number of Nested Classes

- **Definition:**

Counts all classes declared inside other classes.

- **Why it matters:**

It is helpful as it highlights the degree of class coupling and encapsulation, which can affect code readability, maintainability, and testability.

- **Calculation method:**

  Let F be a Java file.

  Let CN be the set of classes whose parent node is of type ClassOrInterfaceDeclaration.

- **Formula:**

NestedClasses(F)=|{c∈F|c.getParentNode()=ClassOrInterfaceDeclaration}|

III.1.4 Number of Declared Interfaces

- **Definition:**

  Counts how many interfaces are declared in the file.

- **Why it matters**

It is helpful as it identifies abstraction layers in the design, providing insight into how well the code adheres to interface-based architecture

- **calculation method:**

  Let F be a Java file.

  Let I be the set of declared interfaces in F.

- **Formula:**

DeclaredInterfaces(F)=|{i∈F|i.isInterface()|

III.1.5 **Number of Abstract Classes**

- **Definition**:
  Counts all class declarations marked with the `abstract` keyword
- **Why it matters**

It is helpful as it reflects the presence of incomplete templates meant for inheritance, which reveals architectural design intentions.

- **Calculation method:**
  Let `F` be a Java file.
  Let `CA` be the set of class declarations marked with the `abstract` modifier.
- **Formula:**

$$AbstractClasses(F) = |\{c \in F | c.isAbstract()\}|$$

III.1.6 **Number of Abstract Methods**

- **Definition**:
  Counts all methods declared `abstract` in classes or all methods in interfaces (which are implicitly abstract).
- **Why it matters:**
  It is helpful as it captures method-level abstraction, which is a key indicator of contract-based design and planned extensibility.
- **Calculation method:**

Let P be the set of all Java source files.
Let MA be the set of all abstract methods found in:
Interfaces (all methods are abstract by default),
Abstract classes (methods explicitly marked as abstract).

- **Formula:**

$$AbstractMethods(P) = \sum_{F \in P} |Abstract\ methods\ in\ F|$$

III.1.7 **Number of Implemented Interfaces**

- **Definition**:
  Counts the total number of interfaces implemented by concrete classes in the file.
- **Why it matters:**

It is helpful as it shows the degree of adherence to abstraction, and how many external contracts the class commits to fulfill.

- **Calculation method:**
  Let F be a Java file.
  Let II be the set of interfaces implemented by classes, retrieved via getImplementedTypes().
- **Formula:**

$$ImplementedInterfaces(F) = \sum_{c \in F} |c.getImplementedTypes()|$$

III.2 Import Conflicts Metric (IC):

The goal of this metric is to evaluate the structure, clarity, and reliability of import declarations, while also identifying common issues such as unused imports, wildcard usage, duplicate imports, and import conflicts.

III.2.1 Metric Categories

The tool analyzes imports and class usage in Java code and provides the following values:

III.2.1.A Total Imports (ICT):

It represents the overall number of import statements in the file.

- Critique / Why it was kept:
  It is helpful as it serves as a baseline for further analysis.

- Interpretation
  A High ICT could suggest that the file may be too large, or that the class has many external dependencies. On the other hand, a low ICT indicates a smaller more specialized class with fewer dependencies.

III.2.1.B Used Imports (ICU):

The number of Imports that correspond to classes or types actually used in the code.

- Critique/Why it was kept:
  It helps at identifying active dependencies.

- Interpretation:
  An elevated ICU may imply that there are many implemented dependencies, and good code maintainability, while a low ICU relative to total imports indicates unused or outdated imports.

III.2.1.C Unused Imports (ICUN):

Number of Imports that are declared but not used anywhere in the code. Helps identify dead or residual imports that can be safely eliminated.

- Critique:
  Helps in identifying dead or leftover imports that can be safely removed.

- Interpretation:
  A large ICUN shows poor maintenance and leftover code that wasn't cleaned up. A lower ICUN shows that the code is clean and reflects good coding habits.

III.2.1.D **Import Conflicts (ICC):**

The Number of imports that reference classes with the same simple name but originate from different packages. It detects ambiguities that could lead to misinterpretation or incorrect type resolution.

Import conflicts are counted based on the number of **distinct class name clashes**. If several classes from different packages share the same simple name, they are treated as a **single conflict** entry, regardless of how many imports are involved. This focuses on the actual risk of ambiguity in code, rather than inflating the count based on how many conflicting imports exist

Ex: Import java.sql.Date;
Import java.util.Date;
Import myPackage.Date; ***This will represent one conflict entry.***

- Critique/Why it was kept:
  Detects ambiguities that could lead to misinterpretation or incorrect type resolution.

- Interpretation:
  A high ICC signals ambiguity and risky design, while a lower ICC value represents safe code with clear class references.

III.2.1.E **Duplicate Imports (ICD):**

Number of identical import statements (same package and class). It represents the total number of repeated import statements, without specifying how many times each individual import is duplicated.

Ex: Import java.util.*;
import java.util.*;
import java.util.*; ***this will count as one duplicate entry***

- Critique/ Why it was kept:
  Indicates redundancy and a lack of attention to detail in import management. May suggest code duplication or carelessness.

- Interpretation:
  High ICC indicates redundancy, rushed development, and lack of review.

III.2.2 **Enhancing IC with Additional Metrics**

The original matric focused on five main metrics: total imports, used imports, unused imports, duplicate imports and import conflicts. While this provides a solid foundation, we can improve it by adding these categories:

III.2.2.A **Wildcard Imports (ICW):**

Number of imports using * to include all classes from a package.

- Interpretation:
  High ICW signals a higher risk of hidden dependencies and low code clarity, and makes for a harder type resolution. Low ICU reflects a more maintainable code and more conventional coding.

III.2.2.B **Unjudged Imports (ICUJ):**

Number of Imports that are part of a name conflict and cannot be definitively labeled as used or unused.

- Interpretation:
  High ICUJ signals the need for manual inspection and more severe import conflict

III.2.3 **Contribution to Fault Prediction**

Import metrics can provide valuable insights into modules that are more likely to contain bugs. Here's how:

- **Code Smell Indicators:** Unused and duplicate imports suggest low-quality or hastily written code, which often correlates with higher defect density.
- **Increased Ambiguity & Risk:** Wildcard imports and import conflicts make it harder for developers (and tools) to resolve types correctly, increasing the risk of semantic errors and maintenance bugs.
- **Maintenance Complexity:** Unjudged and duplicate imports can make code harder to refactor and audit, contributing to technical debt and higher fault-proneness over time.

III.2.4 **Core Logic of the Metric**

The IC metric operates by analyzing the imported classes and the actual classes used in the code. The process follows these steps:

1. **Class Usage Collection**
   All class names used in the code are collected. This includes class references in object creation, variable declarations, method return types, parameters, inheritance, and more.

2. **Import Collection**:
   All import statements in the source file are gathered, including explicit and wildcard imports.

3. **Conflict Detection**:
   If multiple imports bring in classes with the **same simple name** from **different packages**, that class name is flagged as a **conflict**. These conflicts are tracked by name, not by the number of imports, to avoid inflating the metric and to better represent fault risk.

4. **Import Classification**:
   Each import is analyzed to determine its status:

   - If the imported class **is not** involved in a conflict:
   - If its name appears in the used class list then it is used.
   - Otherwise it is unused.
   - If the imported class is involved in a conflict:
   - Its status cannot be classified reliably, so it is marked as **unjudged**.

This classification approach allows for precise categorization of imports and supports deeper analysis of risky or unclear import usage.

III.3 Enhanced OOMR Metric

- **Definition**

OOMR (Overload and Override Method Ratio) is defined by calculating:

1. **TotalMethod**: Total number of methods in a class.
2. **RedefinedMethod**: Number of overridden methods.
3. **OverloadedMethod**: Number of overloaded methods.

The main goal of this metric is to detect anomalies or potentially problematic programming practices related to inheritance and method overloading, which could indicate design flaws or complicate code maintenance.

The metric is calculated as follows:

OOMR = (OverloadedMethod + RedefinedMethod) / TotalMethod

III.3.1 OOMR Metric

III.3.1.A Inheritance Depth Analysis

- Measures how deep a class is in the inheritance hierarchy (all its ancestors).
- High depth may mean more complexity and tight coupling.

III.3.1.B **Override Detection**

- Checks if methods override inherited ones using:
  - @Override annotation
  - Name and signature matching
  - Special checks for equals(), hashCode(), toString()
- Detects problems like missing @Override, empty overrides, or overriding equals() without hashCode() (which breaks Java rules).

III.3.1.C **Overload Analysis**

- Finds methods with the same name but different parameters (overloads).
- Flags issues like:
  - Too many overloads (more than 3)
  - Ambiguous overloads where parameter types cause confusion (e.g., int vs Integer)
  - Generic method names with many overloads hurting readability.

III.3.1.D **Composite Metrics**

- Calculates:
  - Total methods (declared + inherited - overridden)
  - Overload ratio (overloaded methods / total methods), showing API complexity.
- Advanced metrics:
  - **Consistent Extension Rate:** How often overridden methods properly call their parent method using `super`.
  - **Override/Overload Confusion:** When methods are both overridden and overloaded, which can be confusing.

III.3.1.E **Inheritance-Related Code Smells Detection**

- Measures:
  - Override ratio (overridden / total methods)
  - Overload ratio (overloaded / total methods)
- Raises alerts if:
  - Over 70% of methods are overridden (too much inheritance)
  - Overloads dominate (confusing API)
- Checks missing overrides that should come together, like equals() and hashCode(), or toString() for debugging.

III.3.2 **Output, Interpretation & Key Indicators**

III.3.2.A **Design Issue Indicators**

Detects duplicate methods within the same class, which is a sign of poor structure.

1. Class Overview

- Class name (fully qualified)
- List of ancestor classes and interfaces
- Inheritance depth (DIT)

2. Method Inventory

- Declared and inherited method count
- Number of overridden and overloaded methods
- Overload ratio: `Overloaded / Total Methods`

Anomaly Detection Includes

- Overrides missing `@Override`
- Empty method redefinitions
- Excessive overloads (more than 3 per method name)
- Ambiguous overloads (e.g., `int` vs `Integer`)
- `equals()` redefined without `hashCode()`
- Duplicated methods in the same class

III.3.3 Key Formulas

- **Inheritance Depth:**
  `DIT(C) = |A(C)|` → Number of ancestors
- **Total Available Methods:**
  `M_total(C) = |Declared| + (|Inherited| – |Overridden|)`
- **Override Ratio:**
  `OR(C) = |Overridden| / |Inherited|`
- **Overload Ratio:**
  `OLR(C) = |Overloads| / (|Declared| + |Inherited| – |Overridden|)`
- **Overrides without Annotation:**
  Count of overridden methods missing `@Override`
- **Empty Redefinitions:**
  Count of overrides with empty method bodies
- **Excessive Overloads:**
  More than 3 overloads for a method name
- **Ambiguous Overloads:**
  Overloads with similar parameters (e.g., `int` vs `Integer`, similar types, or generics)
- **Override Ratio :**

Quantifies the proportion of inherited methods that are overridden:

OR(C) = |M_ovr(C)| / |M_inh(C)|

- ✓ **M(C)**: Set of methods declared in the class
- ✓ **M_inh(C)**: Set of methods inherited by C from all ancestors
- ✓ **M_ovr(C)**: Set of methods in C that override inherited methods
- ✓ **M_ovl(C)**: Set of methods in C that are overloads of other methods in C

- **Overload Ratio :**

Measures the proportion of methods that are overloads relative to the total available methods:

OLR(C) = |M_ovl(C)| / (|M(C)| + |M_inh(C)| − |M_ovr(C)|)

III.3.4 Detection Logic

- **Override Detection:**
  A method is considered an override if it:
  - Has the `@Override` annotation
  - Matches a method in a parent class
  - Matches a standard override (e.g., `equals`, `hashCode`, `toString`)
- **Overload Grouping:**
  Groups methods by name, then:
  - Confirms signature differences
  - Flags ambiguous pairs based on parameter similarity

III.4 Encapsulation and LCOM5 Metric

- **Overview**

This tool analyzes Java classes by measuring two key object-oriented metrics: **Encapsulation** and **LCOM5 (Lack of Cohesion in Methods, version 5)**. These metrics provide valuable insight into the **quality, structure, and maintainability** of classes, especially in large or evolving codebases.

The application parses Java source files using the **JavaParser** library, then evaluates:

- How well class data is hidden through access modifiers (Encapsulation)
- How closely methods interact with class attributes (LCOM5)

III.4.1 Encapsulation Metric

III.4.1.A What Is Encapsulation?

Encapsulation is the practice of restricting access to an object's internal state using access modifiers like `private`, `protected`, `public`, and package-private (default). It is fundamental to **modular, secure, and robust** object-oriented design.

III.4.1.B Why It Matters

- Protects internal data from external modification
- Encourages abstraction and clear API design
- Makes code easier to maintain and test

III.4.1.C Formula

`pub`, `pri`, `pro`, and `def` are the number of fields with each access level:

```
Total_Attributes = pub + pri + pro + def
Public Rate (%)         = (pub / Total_Attributes) × 100
Private Rate (%)        = (pri / Total_Attributes) × 100
Protected Rate (%)      = (pro / Total_Attributes) × 100
Package-Private Rate    = (def / Total_Attributes) × 100
```

These percentages reveal how much of a class's internal state is exposed versus protected.

III.4.2 LCOM5 Metric (Lack of Cohesion in Methods)

III.4.2.A What Is LCOM5?

LCOM5 measures how cohesive a class is — that is, how consistently its methods use its attributes. High cohesion indicates a well-structured class with clear responsibility; low cohesion may suggest bloated, unfocused code.

III.4.2.B Why It Matters

- Reveals potential violations of the Single Responsibility Principle
- Helps detect "God classes" that do too much
- Encourages clean, modular design

III.4.2.C **Formula**

```
LCOM5 = 1 - (Σ MF / (M × A))
where
```

- `M` be the number of methods
- `A` be the number of attributes
- `MF` be the number of methods that access each attribute

III.4.2.D **Interpretation:**

- LCOM5 ≈ 0 → high cohesion (good)
- LCOM5 ≈ 1 → low cohesion (bad)
- LCOM5 = -1 → not applicable (e.g., no methods or attributes)

III.4.3 **Why These Metrics Were Chosen**

Encapsulation and LCOM5 were selected because they represent two complementary dimensions of object-oriented design:

- **Encapsulation** checks whether classes hide their internal data properly — an essential aspect of good API design and software security.
- **LCOM5** examines how well a class's internal parts work together — a key measure of clarity and maintainability.

Together, these metrics provide a balanced view of **external visibility** and **internal cohesion**.

They are particularly well-suited for automated static analysis because they:

- Are simple to compute
- Offer immediate, interpretable results
- Highlight actionable problems like poor modularity or excessive exposure

By integrating these metrics, the tool supports developers in writing **cleaner, safer, and more maintainable** object-oriented code.

III.5 **Coupling Between Object Classes Metric (CBO)**

The goal of this metric is to assess the degree to which a class depends on other classes. It measures the number of distinct external classes that a given class is using. It calculates:

III.5.1 **Coupling Between Object Classes (CBO):**

The number of unique external classes that a class directly depends on through method calls, field accesses, inheritance, or interface implementation, excluding classes that come from built in java packages like "java.util".

- **Interpretation**:
  High CBO indicates that the class is strongly dependent on other classes, which makes it hard to test and maintain, it also suggests high risk of faults as it is more prone to breakage when other classes change.
- **Formula:**
  JREC = set of all classes of JRE
  CC = set of all classes used in code
  CBO = |CC – (CC ∩JREC)|

III.5.2 **Contribution to Fault Prediction**

Coupling is a well-established indicator of **fault-proneness** in software. Here's how the CBO metric supports fault prediction:

- **Change Sensitivity**: Highly coupled classes are more likely to break when related classes change.
- **Code Complexity**: High CBO suggests the class has more responsibilities or is doing too much, which increases the chance of logic errors.
- **Testing Difficulty**: Tightly coupled classes are harder to isolate for unit testing, reducing test coverage and increasing undetected bugs.

III.6 **Number of Children (NOC) Metric**

III.6.1 **Definition of NOC (Number of Children):**

NOC (Number of Children) measures how many direct subclasses inherit from a given parent class in an object-oriented hierarchy. This metric helps evaluate:

- Class importance in the inheritance tree
- Potential design issues (overly complex hierarchies)
- Impact of changes to parent classes

III.6.2 **Implementation Overview**

The ChildrenNB class analyzes Java source code to calculate NOC:

Key Components

I. **Input Processing**
   - Takes path to a Java class file
   - Parses the file using JavaParser
   - Extracts base class name and source folder location

II. **Recursive Scanning**
   - Scans all .java files in the source directory
   - Identifies classes extending the base class
   - Maintains count and names of extending classes

III. **Analysis Features**
   - Handles nested packages
   - Skips final/private classes (non-extensible)
   - Collects fully-qualified names of child classes

III.6.3 **Impact on Error Detection :**

- **High NOC**:
  - Increases maintenance complexity
  - Makes parent class changes riskier
  - May indicate violation of Single Responsibility Principle
- **Low NOC**:
  - Suggests limited reuse potential
  - May indicate missed abstraction opportunities

III.7 Exception Type Classification (ETC)

III.7.1 Definition of ETC (Exception Type Classification):

The **ETC metric** is a quantitative analysis tool that classifies exceptions in Java code into distinct categories to evaluate error-handling quality. It identifies:

- **Total exceptions** (thrown, caught, or declared).
- **JDK built-in exceptions** (standard Java exceptions).
- **Custom exceptions** (user-defined exceptions).
- **Jdk Checked exceptions** (must be handled or declared).
- **Jdk Runtime (unchecked) exceptions** (do not require explicit handling).
- **Jdk Errors** (critical system failures).

This classification helps developers detect poor exception-handling practices that may lead to software defects.

III.7.2 Importance in Error Detection

- **The ETC metric helps in**:
  - Identifying Unhandled Exceptions: Highlights exceptions that are thrown but not caught.
  - Evaluating Defensive Programming: A high number of unchecked exceptions may suggest missing input validation.
  - Assessing Code Robustness: Excessive use of custom exceptions may indicate over-engineering or tightly coupled error handling.
  - Detecting Critical Failures: Presence of Error types can signal system-level instability.

III.7.3 How it works :

the metriq work with 3 classes :

a. **Exceptionscounter (Data Collector)**

- Role: Extracts exceptions from source code
- Key Methods:
  - getThrownExceptions(): Finds all throw new statements
  - getCaughtExceptions(): Identifies catch block exceptions
  - getDeclaredThrownExceptions(): Checks method throws clauses
  - mergeExceptionSets(): Combines all found exceptions

b. **ListJDKClasses (JDK Exception Database)**

- **Role**: Identifies standard Java exceptions (by dearching in the src.zip in the jdk file that is being use currently by eclipse)
- **Key Methods**:
  - jdkclasses1(): Lists core JDK exceptions
  - jdkclasses2(): Lists sun./*com.sun.* exceptions

Remark:

In this implementation, we chose to explicitly list **JDK exception classes** to ensure precise classification and broad applicability. While frameworks like **Defects4J** encapsulate projects that use their own internal exceptions and dependencies, this is not always the case in general Java applications. In real-world systems, **codebases may span multiple modules or external libraries**, and **intersections between two Java projects** can lead to ambiguity if exceptions aren't clearly distinguished. Therefore, relying on an explicit JDK class list improves reliability and consistency across diverse environments.

c. **ExceptionType (Main Class)**

- Role: Coordinates exception analysis
- Key Methods:
    - ExceptionType(String path): Initializes analysis by:
        1. Creating Exceptionscounter instance
        2. Calculating total exceptions (TNE)
        3. Classifying exceptions (JDE, CEC, etc.)

III.7.4 Outputs of ETC and Their Role in Error Detection

III.7.4.A **TNE (Total Number of Exceptions) :**

it represent the number of times exceptions are used in a given class ,

it represents : TNE = ThrownExceptions + CaughtExceptions + DeclaredExceptions

- Impact on Error Detection :

A high TNE suggests complex error-handling logic, increasing defect likelihood.

High TNE → Many error-handling paths → Potential complexity

Low TNE → Possibly missing important checks

III.7.4.B **JEC (JDK Exception Count)) :**

it represent the number of times build in exceptions are used in a given class.

Check if exception is in ListJDKClasses.jdkclasses1() or jdkclasses2().

Formule : JEC = TNE – CEC

- **Impact on Error Detection :**

A high number of JDK exceptions (like IOException, NullPointerException, etc.) shows that the code relies heavily on built-in mechanisms to report common problems. It tells you what kind of problems the program expects.

III.7.4.C **CEC (Custom Exception Count)**

Number of user-defined exceptions.

Formule : CEC = TNE – JEC

- **Impact on Error Detection :**

High CEC → Possible good domain-specific design, or over-complexity

Custom exceptions may be more prone to bugs due to lack of standardization

III.7.4.D **JCKE (JDK Checked Exception Count)**

the number of build in Exceptions requiring explicit handling

to define it : JCKE = | instanceof Exception && !instanceof RuntimeException |

- **Impact on Error Detection :**

High JCKE → Code is likely to be more robust and prepared for recoverable errors

Low JCKE → Risk of missing edge-case handling

III.7.4.E **RTE (Runtime Exception Count)**

the number of build in Exceptions that does't need explicit handling

formule : **RTE** = | instanceof RuntimeException |

- **Impact on Error Detection :**

High RTE → May indicate poor defensive coding (e.g., NullPointerException, IndexOutOfBoundsException)

Low RTE → Indicates better validation and safer code

III.7.4.F **ERR (Error Count)**

The total number of Java Error used, thrown, or caught in a Java file. This calculate only classes that extend java.lang.Error (not Exception), such as OutOfMemoryError, StackOverflowError, or AssertionError.

- **Impact on Error Detection :**

ERR > 0 → Presence of critical system-level failures

May indicate unstable or dangerous operations in the program

IV. Application Screenshots

Figure 1 Home Page

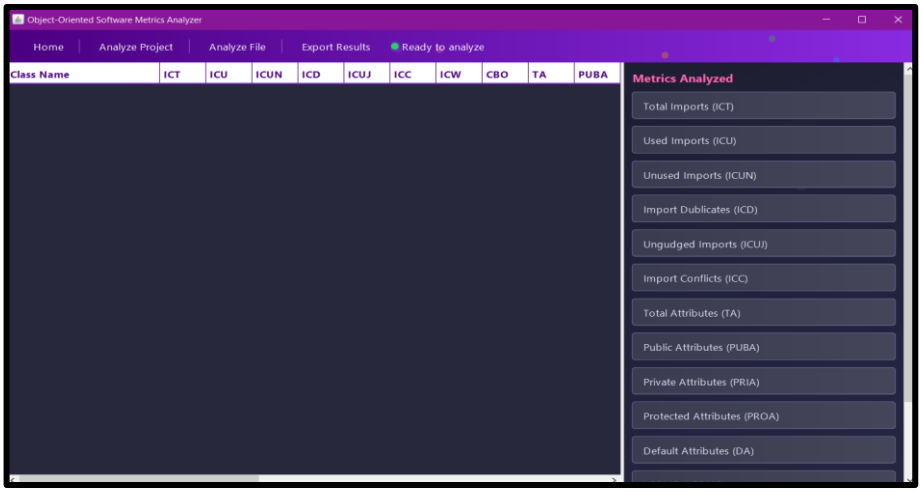

Figure 2 Analysis Page

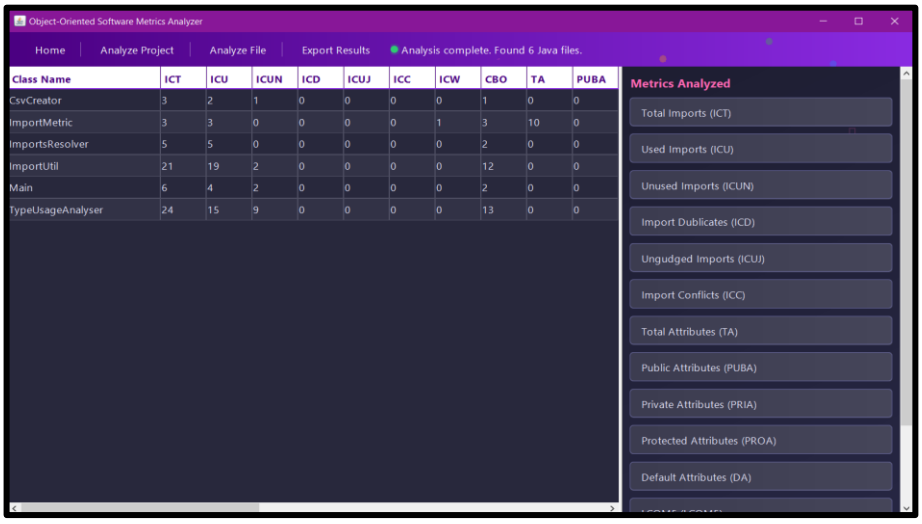

Figure 3 Analysis Results Page

Conclusion

This project was developed with the goal of making software quality assessment more practical and accessible. By analyzing object-oriented metrics in real-world Java code from the Defects4J dataset, we've built a tool that brings valuable insights into code structure, complexity, and maintainability. Using libraries like JavaParser to traverse the Abstract Syntax Tree, the application performs deep analysis while remaining fully automated.

The interface, developed using Java Swing, focuses on clarity and usability—providing an interactive environment where users can view and export results with ease. Backed by an MVC architecture, the system is organized, maintainable, and ready for future enhancements.

Looking ahead, this work opens the door to integrating AI models for defect prediction. With a rich set of structured metrics already in place, the system can be extended to train and evaluate machine learning algorithms, potentially enabling automated identification of code likely to contain defects. This adds a new layer of intelligence to the tool—shifting from descriptive analysis to predictive capabilities.

At its core, this project bridges theory and practice. It offers developers and researchers a simple yet powerful way to understand how object-oriented principles affect code quality, and how measurable metrics—combined with AI—can support better software design and early defect detection.

Resources and References

- **Supervisor Consultations**

Our advisor gave us valuable input on both technical direction and implementation strategy through regular check-ins.

- **Open-Source Repositories**

Public GitHub projects served as test cases for metric analysis and defect detection. They helped validate the real-world relevance of our approach.

- **Reference Applications**

We took design cues from existing applications when it came to layout, styling, and architecture — especially those following modular and MVC principles.

- **YouTube Tutorials**

Helped us with specific Swing components, UI design tricks, and general application styling.

- **ChatGPT and Other AI Tools**

Used for brainstorming fast research and explanation and even designing visual elements like the app logo.

- **Academic Resources**

Books, PDFs, and articles on object-oriented metrics, software quality, and static analysis helped us stay grounded in solid theory

- Chidamber, S. R., & Kemerer, C. F. (1994). A Metrics Suite for Object Oriented Design
- IEEE Transactions on Software Engineering.
- An Overview of Various Object Oriented Metrics, By Brij Mohan Goel & Prof. Pradeep Kumar Bhatia, International Journal of Information Technology & Systems, Vol.
- Applying and Interpreting Object Oriented Metrics, By Dr. Linda H. Rosenberg :Track 7 – Measures/Metrics.
- Metrics For Object Oriented Design (MOOD) To Asses Java Programs ,Prof. JUBAIR J. AL-JA'AFER & KHAIR EDDIN M. SABRI, University of Jordan