New metrics for object-oriented software based on regular expression

Fatma Zohra Mekahlia[1,2,*], Rabah CHABANE-CHAOUCHE[2]

[1]MOVEP Laboratory.

[2]Faculty of Computer Science, University of Science and Technology Houari-Boumédiène Bab Ezzouar, Algiers, Algeria.

Abstract

In software engineering, the testing is a crucial phase of the software development life cycle. which represents a specific sequence of activities ensuring that software quality objectives are achieved. Currently, development companies have the heavy task of performing different types of testing which takes a lot of time and effort. Therefore, software defect prediction becomes a hot topic that allows estimating the parts of the code that are prone to failures in order to allow testing on these estimated parts only. Some researchers have proposed the use of software metrics to describe the characteristics of software evolution. For this reason, we have proposed six new object-oriented software metrics that monitor: import conflicts, exception conflicts, encapsulation rate, overload and overrid method ration, number of each Swing component. Furthermore, we have formalized these metrics using regular expression with lookaround assertions. This article essentially aims to: 1- study the software fault prediction procedure. 2-propose new metrics to strengthen the prediction. 3- formally proved the new metrics using regular expressions.

Keywords

Software metrics, object-oriented, regular expression, Java, lookaround assertions, software engineering, formal model,

1. Introduction

1.1. Motivation

In the field of software defect prediction, several works have shown that there is a strong relationship between the number and quality of object-oriented metrics which influence the detection of subject classes such as: [1] [2] [3] [4] [5] [6] [7] [8]. Since the detection and determination of software defects is a very delicate, difficult and costly phase compared to developers who spend a lot of time to target and correct them in the different phases of the software life cycle such as unit tests, functional tests, integration tests, etc. The transition to software defect prediction becomes essential because it allows to reduce the effort, time and resources to identify any type of software defects before the delivery of the product to the end customer. Generally, software defect prediction is based on machine learning models [9] [10] and software metrics [11] [12]. Which makes the emergence and development of new software metrics play a very important role in the quality and accuracy of software defect prediction [13] and what prompted us to study the paradigm of object-oriented programming to propose new metrics that represent fundamental concepts of OOP and that positively influence the prediction of software defects. Existing solutions suffer from lack of certain metrics that represent fundamental concepts in object-oriented software such as:

- Import conflicts of classes in a program. The development tools of the object-oriented software that exist do not control the imports of a project which can generate conflict errorscompared to general import (*).
- The absence of measures that evaluate the degree of encapsulation in classes, knowing that encapsulation is a fundamental principle of object-oriented programming. The evaluation of the degree of encapsulation helps to avoid providing other undesirable side effects such as data manipulation errors or unintentional external interferences.
- Failure to measure the degree of method redefinition and overloading that could cause common problems for developers. Indeed, overuse of these mechanisms can lead to excessive complexity and compromise the modularity of the code.
- Failure to measure the degree of use of exceptions in a program that can help detect potential security risks.
- The failure to measure the complexity of the GUI user in terms of the use of the swing graphics library because excessive use of certain components can negatively affect the performance of the software due to the large number of swing components used.

Our goal is to increase the set of object-oriented software metrics by addressing the gap we have targeted in order to increase the performance as well as the accuracy of predicting software defects.

Regular expressions are a formalization of John von Neumann's automata theory that was later formalized by the mathematician Stephen Kleene. Afterwards, regular expressions were first implemented by Ken Thompson in a software with a patent on the proper use of pattern matching and of course this formalism has been reimplemented in many ways up to the present day. Today regular expressions can be applied to extract information from text at a very high level and we can say that regular expressions have exceeded the traditional mathematical limits. However, we can use them to search for tokens or particular elements in a large text. For example, detecting email addresses in a text editor in order to identify spam emails. Also, they can be used to replace target tokens in a text by other particular elements in the case of data cleaning for example. Generally, in arithmetic, we use operations such as * and / to construct expressions. As example: 5*4/2. In the same way, we can use regular operations to construct regular expressions while describing a regular language, called regular expressions. For example: $(a \bigcup b)^*$.

The 4th International Conference on Trends and Advances in Collaborative applied Computing, Nov 25–26, 2024, Constantine, Algeria.

*Corresponding author.

mekahlia.fzohra@yahoo.fr (F. Z. Mekahlia);

chabanechaoucherabah4@gmail.com (R. CHABANE-CHAOUCHE)

0000-0002-7369-3878 (F. Z. Mekahlia)

1.2. Contributions

This work consists of the proposal and formal validation of new object-oriented metrics which will help in the future to predict object-oriented software defects. The main contributions of this paper are summarized below:

1. Proposal of six new object-oriented metrics that allow to measure the following aspects of a software: import conflicts, degree of encapsulation, overridden and overloaded methods, potential risk generated by the use of exceptions and complexity of the graphical user interface and other performance measures like: file size, non-empty lines, empty lines, comment lines, total number of lines.

2. We have formalized our proposed metrics using regular expressions lookarounds in Java where we have presented our mathematical semantics.

To our knowledge, this work is the first work that proposes new object-oriented metrics that measure fundamental concepts in object-oriented programming such as method redefinition and overloading, complexity of the graphical user interface with formalization of the metrics. Generally, the existing works have proposed metrics that manage the general aspect of a software such as the number of lines of code in addition the metrics have been presented by a simple formula.

The paper is organized as follows. Related work was presented in Section 2. In section 3, we will present our discussion on related work. In Section 4, formal semantics have been presented and we will end with a conclusion and future work in the section 5.

2. Related works

Since the emergence of the object-oriented paradigm in 1990s, several software metrics have been proposed in the literature that allow to evaluate the characteristics of an O.O software. In [14], the authors invented a suite of metrics that allow you to measure the complexity of the object-oriented code. Commonly called CK metrics are still references today. The metrics were validated using commercial systems written in C++ and SmallItalk. Subsequently, in [15] the authors propose a data model and terminology with illustration of the importance of CK: 1) Weighted Methods per Class, 2) Depth of Inheritance Tree, 3) Number Of Children, 4) Coupling Between Objects, 5) Response For a Class, 6) Lack of Cohesion in Methods.

In [16] we find the MOOD metrics (Metrics for Object Oriented Design) which measure certain characteristics of object oriented such as Encapsulation, Coupling, Inheritance and Polymorphism. In [17]the authors present their tool, which is based on MOOD metrics proposed: 1) Method Hidden Factor, 2) Attribute Hidden Factor, 3) Method Inheritance Factor, 4) Attribute Inheritance Factor, 5) Polymorphism Factor, 6) Coupling factor.

In [18] the authors presented through their surveys a comparative study between proprietary, free and open source tools that evaluate static and dynamic object-oriented metrics. Moreover, the metrics supported in dynamic metrics calculation tools are very limited such as inheritance, dynamic binding and runtime polymorphism.

The authors in [19] investigated the relationship between centrality measures and O.O-metrics in order to predict the propensity for failure in three aspects: fault-prone classes, fault severity and number of faults. As a conclusion, he finds that the use of O.O-metrics and centrality measures improves the prediction of fault-prone classes and the number of failures in a program. To carry out this study, the authors focus on 9 metrics that belong to three families: complexity, coupling, and size.

Regular expressions, also called regex [20], are formal patterns that allow the extraction of tokens or target character strings from a text or a computer program. A regular expression is represented by a set of metacharacters where each metacharacter represents a given meaning. For example ([ ]) : Match any one of the enclosed characters. Example: [abc] matches 'a', 'b', or 'c'. In the literature, we find several works which focus on the use of regular expressions as a formal solution which makes it possible to target a certain text or program code in order to carry out certain processing on the targeted area. In [21] the authors proposed the FungiRegEx software which is based on regular expressions and which allows proteomic research. The software retrieves real-time data on several species from the JGI Mycocosm database.

3. Discussion

According to what we have observed in the work carried out, the quality of software metrics plays a very important role in the prediction of software defects and makes it possible to increase the prediction performance. For this reason and in this work, we aim to propose new object-oriented metrics in order to improve the performance of software defect prediction. Furthermore, our second objective consists of formally proving using regular expressions the validity of our new metrics which will be used in the future to strengthen the prediction of software defects.

4. Proposed software metrics

Developing computer software requires a methodical and rigorous approach, especially when it comes to measuring the quality and performance of the code. In this context, the evaluation of metrics plays a crucial role. A metric, in the field of software engineering, is a quantitative measure used to evaluate various aspects of a software system, including its complexity, quality, maintainability, and performance. Thus, this process allows informed decisions to be made throughout the development cycle, from design to implementation, including validation and maintenance. Therefore, we followed several key steps to evaluate and propose six new object-oriented metrics specifically for the Java language in order to improve the quality and efficiency of object-oriented software, which are:

4.1. Import Conflict (IC)

Import conflicts often occur during software development. However, available Java development tools do not provide a comprehensive analysis on this topic. Therefore, we proposed the IC metric to fill this gap. This metric helps to manage the import of a Java file with all possible scenarios, it provides insights into the complexity and class dependencies. IC retrieves all the imports in a special file and then classifies them into four distinct categories which are:

- Used Imports: the import is used at least once. Example: import java.util.*. We propose to modify the import by using only the class concerned.

- Not Used Imports: the import is never used. We propose to delete the import.
- Duplicate Imports: if the import is repeated at least once. we propose to remove the painful copies.
- Conflict Imports: two or more imports have one class in common. Example: import java.sql.date and import java.util.date. We propose to delete one of the two.

Impact in software engineering:

- A large number of imports can increase code complexity and extend compilation time, thus affecting software maintainability and performance.
- Better visibility of imports in a class improves its understandability and maintainability, reducing unnecessary code complexity.
- This metric gives insight into a project's imports and helps in detecting conflict errors even in the general import case (*) that the IDE does not detect.

4.2. Java Exception Analyser (JEA)

Help a detect potential security risks and strengthen the overall software security by evaluating this aspect of the code. This metric analyzes the Java project to extract exceptions and classifies them into two groups: default Java exceptions and non-default exceptions (exception made by the user), and itself divided into two subsections: 1. Runtime exception: exception that extends RunTimeException. 2. Compile time exception: exception that doesn't extends RunTimeException.

Impact in software engineering:

- A high number of exceptions handled indicates attention to error handling and exceptional cases.
- A large number of potential exceptions can indicate excessive complexity in the code, which can make the code more difficult to understand, maintain, and debug.
- The metric is designed to improve code quality assessment by providing information about exception handling practices.

4.3. Encapsulation Rate (ER)

Encapsulation is one of the fundamental principles of object-oriented programming and a crucial aspect of code quality. Therefore, we have proposed the ER metric to better evaluate and monitor this concept. It measures the degree of encapsulation of a class and protected from direct access from other parts of the program. Fetches number of members within a Java file (method, class) and categorize them by their access modifiers: public, package friendly, protected and private.

In addition we calculate the ratio of encapsulation using the below formula, where $|Private|$ and $|Protected|$ represent the number of private and protected members respectively, and $Total$ represents the total number of members:

$$\text{Ratio of encapsula} = \frac{|Private| + |Protected|}{Total} \quad (1)$$

Impact in software engineering:
Measuring the degree of data encapsulation of a class can be used to evaluate the design quality of a class in an system. A class with a high encapsulation rate is generally considered to be better designed because it promotes modularity, reusability, and code maintainability. On the other hand, a class with a low encapsulation rate may be prone to undesirable side effects, such as data manipulation errors or unintended external interference. Thus, by monitoring and optimizing the encapsulation rate of classes, developers can improve the robustness and reliability of their software.

4.4. Overload and Overrid Method Ratio (OOMR)

Most developers want to identify areas in the code where inheritance and method overloading are overused, as overuse of these mechanisms can lead to excessive complexity and compromise the modularity of the code. Therefore, we proposed a specific metric to evaluate and monitor this aspect. Our system retrieves methods of a class for categories in two parts: override methods et overload methods.

Then it calculates the ratio of overrided and overloaded methods and the OOMR ratio using the below formulas , where $|Overload|$ and $|Override|$ represent the number of overload and overload methods respectively, and $Total$ represents the total number of methods:

$$\text{Override Method Ratio} = \frac{|Override|}{Total} \quad (2)$$

$$\text{Overload Method Ratio} = \frac{|Overload|}{Total} \quad (3)$$

$$\text{OOMR Ratio} = |Overload| + |Override| \quad (4)$$

Impact in software engineering:
This proposed metric can be useful to assess the complexity and maintainability of a software system. Thus, by calculating the OOMR, we can obtain indications on the quality of the design and identify potential areas requiring revision or optimization in the code. As an example, a high OOMR indicates an overuse of inheritance mechanisms, which can make the code more difficult to understand and maintain. On the other hand, a low OOMR indicates good object-oriented design with appropriate use of inheritance and method overloading. Finally, this metric offers insight into code reuse and the complexity of the object-oriented design, and allows to obtain valuable information on how methods are manipulated in the code, thus facilitating the analysis of modularity.

4.5. JavalyzerX (JAX)

Understanding a Java project is an essential step for its improvement and development, given the lack of a tool that meets all these criteria in one go. We proposed a metric that gives a complete analysis of the project that will serve as a balance sheet, encompassing various aspects and providing valuable insights into the structure, size and performance of the code. The JAX metric is a composite static metric that includes several sub-metrics. The analysis includes metrics such as:
1. Number of line: fetches total number of line and categorize them in two section:

- **Code statistic: a.** Number of empty line and number of code line. **b.** Number of curly braces only line (has only '{' or '}'). **c.** Ratio of code line using the

formula , where $|Code|$ is number of code line and $|Total|$ is total number of line:

$$\text{Ratio of code line} = \frac{|Code|}{Total} \quad (5)$$

- **Comment statistic: a.** Number of line of comment only. **b.** Ratio of comment line only using the below formula , where $|Comment|$ is number of comment line only and $|Total|$ is total number of line:

$$\text{Ratio of comment} = \frac{|Comment|}{Total} \quad (6)$$

2. Software: a. Method prototype and interfaces implemented. **b.** Classes: Sub classes and abstract classes. **c.** Parent of the Java file initialized by 'Object' in case it isn't extending another class.
3. Performance: a. size of the file in byte. **b**. RunTime of the file in seconds.

Impact in software engineering:
This metric provides a robust solution to analyze Java files in a specified folder. By providing valuable insights into code structure, size, and performance, it improves the understanding and optimization of Java projects.

4.6. Swing Component (SC)

Swing is a widely used graphical library for creating user interfaces in Java applications. Excessive use of some components can negatively affect the application performance. Java applications can become complex due to the large number of Swing components used. In order to enable developers to make informed decisions about the organization and structure of the code, which in turn can help improve the maintainability of the code in the long run, we have proposed a metric that calculates the exact number of each Swing component in the code. SC is a static metric of user interface complexity. It takes the swing elements in the Java file and classifies them according to their type.

Impact in software engineering: this metric provides an accurate assessment of user interface complexity, allowing developers to better understand the workload associated with maintaining and extending it, it also facilitates the optimization of overall application performance by monitoring component usage trends, as it contributes to the effective management of code complexity by allowing developers to make informed decisions about the organization and structure of the code to improve its maintainability.

5. Formal semantics

In this section we will present our mathematical semantics which is based on regular expressions with lookaround to formalize the six metrics proposed in Java. Let r be a regular expression and $\sum$ be an alphabet. The language represented by r denoted L(r) is a regular language. r is a sequence of symbols, like: union, concatenation, alphabet. To present our proposed engine, we will start by defining the syntax of our Java's regex:

5.1. Java's regex syntax

5.1.1. Quantifiers

Quantifiers are symbols for specifying how many times a pattern should appear in a regular expression. Our used quantifiers are:

- *: Matches 0 or more occurrences of the preceding pattern.
- +: Matches 1 or more occurrences.
- ?: Matches 0 or 1 occurrences.
- {n,m}: Matches between n and m occurrences..
- {n,}: Matches n or more occurrences.
- {n}: Matches exactly n occurrences.

5.1.2. Alternation

Alternation is a strong concept in regular expressions which allows you to specify several alternatives for the same pattern:

- Vertical Bar (|): Acts as a logical OR between patterns.
  Example: a | b matches 'a' or 'b'.

5.1.3. Character Classes

Represent sets of characters that describe specific search patterns and which are:

- Square Brackets ([ ]) : Match any one of the enclosed characters.
  Example: [abc] matches 'a', 'b', or 'c'.
- Ranges: Specify a range of characters.
  Example: [a-z] matches any lowercase letter. i.e. match any character that belongs to the specified set
- Negation: Use to negate the character class. i.e. match any character that does not belong to the specified set.
  Example: [0-9] matches any character that is not a digit.

5.2. Escaped characters

In regular expressions, some characters cannot be used directly because they have a special meaning. To use them, we will need to escape them with an escape character (\):

- \d : Matches any digit, equivalent to [0-9].
- \D : Matches any non-digit, equivalent to [0-9].
- \w : Matches any word character (alphanumeric plus underscore), equivalent to [a-zA-Z0-9_].
- \W : Matches any non-word character.
- \s : Matches any whitespace character (spaces, tabs, line breaks).
- \S : Matches any non-whitespace character.
- \n : Matches new line .

5.2.1. Special Characters

- Dot (.) : Matches any single character except newline (\n) .

5.2.2. Lookahead and Lookbehind

Regular expression lookarounds are very practical for checking the presence or absence of a left or right subexpression in relation to the current position while using constraints on the context in the searched pattern [22]. Furthermore, if the regular expression engine tests a lookaround, it does not advance in the text but rather it stays in its place, it can advance if and only if the condition defined in the lookaround

is tested correctly. You should also know that there are two types of lookaround which are lookaheads and lookbehinds.

Lookaheads to test the presence or absence of a pattern ahead of the searched pattern by specifying conditions to check after the current position. On the other hand, lookbehinds to test the presence or absence of a pattern after the searched pattern by specifying conditions to check before the current position.

We also have positive and negative lookaheads. For positive lookahead checks that the expression must be found after the current position and without including this expression in the global match ((?=expression)). On the other hand, negative lookaheads ((?!expression)) checks if the expression is not found after the current position and therefore the opposite.

Definition: Let $\sum$ be an alphabet. Lookaround tests the presence or absence of a pattern just before or just after the searched pattern as follows:

- (`(?=pattern)`): it is positive lookahead. Asserts that what follows the current position in the string matches the pattern inside the parentheses.
  Example: `a(?=b)` matches 'a' only if it is followed by 'b'.
- (`(?!pattern)`): it is negative lookahead asserts that what follows the current position in the string does not match the pattern inside the parentheses.
  Example: `a(?!b)` matches 'a' only if it is not followed by 'b'.
- (`(?<=pattern)`): it is positive lookbehind asserts that what precedes the current position in the string matches the pattern inside the parentheses.
  Example: `(?<=a)b` matches 'b' only if it is preceded by 'a'.
- (`(?<!pattern)`): it is negative lookbehind asserts that what precedes the current position in the string does not match the pattern inside the parentheses.
  Example: `(?<!a)b` matches 'b' only if it is not preceded by 'a'.

Example:
Consider the regular expression \w+(?= email) searches any alphanumeric word that is followed by the word "email". Lookahead allows you to check the presence or not of the word "email" after the current position with non-inclusion in the global correspondence.

5.3. Language proposed

5.3.1. Type Parameter

Type parameter is used to build other regex like the genericity part of class and method detection regex.
Example: public class NumberBox <T extends Number> { public static <T extends Number> double sum(T[ ] array) {}}

L_1 = { Type Parameter Of Genericity }

- MultipleBoundPattern = (\s+extends\s+\w+(\s*<\w+>\s*)?(\s+&\s+\w+ (\s*<\s*\w+\s*>)?)*\s*)?, matches one/multiple bounds in a Type Parameter extends NumberClass & <T >.
- TypeParameterGen = (\s*<\s*\w+(MultipleBoundPattern) \s*(,\s*\w+(MultipleBoundPattern)\s*)*\s*>\s*)?, matches Type Parameter <T extends NumberClass, k extends <V »

5.3.2. Method Detection

- IC(Import Conflict): it's needed to fetch the classes from the method prototype (return type , parameter type , exception thrown) because some of them need imports to be used.
  Example: public static ArrayList <ImportStatus>ImportFetch(File file){...}
- ER(Encapsulation Rate): the metric needs to obtain the method prototype from the java file to fetch its access modifier.
  Example: private void switchButtonToPoly() {...}
- SM(Swing Component): it's needed to fetch the swing element from method prototype (return type , parameter type , exception thrown).
  Example: public JPanel createPanelWithButton(JButton button)
- JAX(Java Analyzer): the metric needs to fetch the prototype method from the java file.
- JEA(Java Exception Analysis): it needs to fetch the exception thrown in a method prototype.
  Example: public static int countOverrideMethods (Class<?>) throws FileNotFoundException{ ...}

L_2={Method Or Constructor Prototype}

- Bracket=(\[\s* \]){1,2} ,to match [ ] or [ ][ ] of arrays.
- ArrayDeclarationPattern=\w+\s+\w+\s*(Bracket) | \w+ \s*(Bracket)\s*\w+ \s*, to match 2d or 1d array declaration like : int[][] IntegrGrid, Student ArrayStudent [].
- ArrayTypePattern = \w+\s*(Bracket)\s*, to match 1d or 2d array type without name of the array example:int [].
- NormalPattern = \w+\s+\w+ ,to match simple type: int nb, float pi.
- WrapperClass = \s*\w+\s*, to match WrapperClass Inside the <>of a collection: Integer.
- WildCardGen = \s* \? (extends\s+ \w+ | super \s+ \w+)?\s*, matches wildcard genericity.
- SimpleInside = \s*(WrapperClass)\s* | \s*(ArrayTypePattern)\s* | (WildCardGen), to match either wrapper class or arrays inside <>of a collection.
- InsideCollection = (SimpleInside) | \s*\w+<\s*(SimpleInside)\s*> | \s*\w+\s*<\s* (SimpleInside)\s*,\s*(SimpleInside)\s*>\s* ,to match double nested or normal inside of a collection.
- SetListPattern = \w+\s*<\s*(InsideCollection)\s*>\s* ,to match set and list collection.
- MapPattern = \w+\s*<\s*(InsideCollection)\s*,\s* (InsideCollection)\s*>\s* ,to match map collection.
- CollectionPatten = (MapPattern) | (SetListPattern).
- Paramter = \s*(NormalPattern)\s* | \s*(Array-DeclarationPattern)\s* | \s*(CollectionPattern) \w+\s* ,to match collection, array and simple type.
- Arg = \(\s*((Paramter)(,\s*(Paramter))*)?\s*\) ,to match the Arguments including the parenthesis of a method it also matches no arguments.
- AcessModifier= (private\s+ | public \s+ | protected \s+)?.
- NonAcessModifierSimple = (static\s+ | final \s+ | abstract\s+)?.

- ModifierSimple = (AcessModifier) (NonAcessModifierSimple) | (NonAcessModifierSimple) (AcessModifier).
- ModifierComplex = (AcessModifier)final \s+static \s+ | (AcessModifier)static\s+final \s+ | final \s+ (AcessModifier)static\s+ | static\s+(AcessModifier) final \s+ | static\s+final\s+(AcessModifier) | final\s+ static\s+(AcessModifier).
- ModifierPattern = ModifierSimple |ModifierComplex.
- ThrowsPattern = (\s*throws\s+\w+\s* (\s*,\s*\w+ \s*)*)?, to match single or multiple exceptions throws.
- CurlyBraces = (\{\s* | {\s*\}\s*)?, to match { or {}.
- ReturnType = \s*\w+\s+ | (collectionPattern) | (MapPattern) | (ArrayTypePattern) ,to match return type could be array like int[ ] , simple type : Integer , collection : List <ArrayList<String []».
- ConstructorRegex=(AcessModfier)(TypeParameterGen) \w+ \s*(Arg)\s*(ThrowsPattern)(CurlyBraces) ,to match constructor prototype : ImportStatus(String ImportName,int ImportStatus,int LineNumber) { ...}.
- MethodPattern = (ModifierPattern) (TypeParameterGen)(RetunType)\w+\s*(Arg) (ThrowsPattern)((CurlyBraces) | \s*;\s*), to match normal method prototype: static Set<String > FetchSrcPackageFile(String AsterixImport){ ...}.
- MethodPrototypePattern= (MethodPattern) | (ConstructorRegex), to match method prototype.

5.3.3. Import Detection

IC(Import Conflict): it's needed to fetch the imports from a Java file.
L_3 = {Import Line}

- StaticAccesModifier = (\s*static\s+)? , to match static access modifier for static import.
- ImportPattern=\s*import\s+(StaticModifier)\w+(\s*\.\s* (* | \w+))+\s*;\s*, to match static and normal import line : import static java.util.math.*; , import application.BackEnd.RegularExpression;

5.3.4. Package Detection

it's there to avoid package statement while reading the Java file line by line since it doest hold any interesting data.
L_4 = {Package Line}

- PackagePattern =\s*package\s+\w+(\s*\.\s*\w+)*\s*;\s*, to catch package line : pacakge application; , package application.Backend;

5.3.5. Catch Detection

- IC(Import Conflict): some exceptions inside catch statement needs to be imported to be used.
  Example: catch (FileNotFoundException | MalformedURLException | ClassNotFoundException e).
- JEA(Java Exception Analysis): metric needs to fetch the exceptions inside a catch statement.
  Example: catch(Exception e)

L_5={Catch Statement}

- OptionalClosingCurlyBraces = (\s*\}\s*)?, to match } before the catch statement.
- SingleCatch = \s*\w+\s+\w+\s*, to match single exception catch: Exception e.
- MultipleCatch = \s*\w+\s*(\s*\ | \s*\ w+\s*)*\s*\ | \s*\w+\s+\w+\s*,to match multiple exception catch : FileNotFound | IOException e.
- InsideCatch = (SingleCatch) | (MultipleCatch), to match what's between the parasyntheses of a catch statement.
- CurlyBraces = (\{\s* | {\s*\}\s*)?, to match optional curly braces in the end of a catch statement : { or { }.
- CatchPattern = \s*(OptionalClosingCurlyBraces)catch\s* \((InsideCatch)\)\s*(CurlyBraces)\s*, to match a catch statement.

5.3.6. String Literal

It's used to build the method call regex since a string literal can be passed as a parameter.
L_6={String Literal with all concatenation possible }

- Char= ["\n]+, matches any characters beside" and newline.
- StringConcatElement= (ClassVariable) | (MethodCall) | \w+ | "(Char)" | (NumbersPattern), matches methodCall: token.getToken(), ClassVariable: Student.Name, VarName: Age, or a String: "Hello World!", Numbers: 21, -32.21f.
- StringConcat = (StringConcatElement)(\+(StringConcatElement))*, this matches one/multiple concatenation with StringConcatElement.
- LiteralStringPattern = ((StringConcat)\+)?"((Char) | " \+(StringConcat)\+ ")"(\+(StringConcat))? , matches string literal with optional concatenation in the beginning middle and end : Age + "Is My Age and My Name is "+ Student.Name + "My Grade Is"+ 13.21f.

5.3.7. Numbers

The numbers regex is used to build the string literal regex since a number can be concatenated with a string literal , it's also used as parameter in method call.
L_7={Int And Float And Double}

- SignPattern = (\+\s* | \-\s*)?, to match sign of numbers: none , + , -
- FloatPattern = \s*(SignPattern)\d+\.\d+(f)?\s*, to match double and float.
- IntPattern = \s*(SignPattern)\d+\s* , to match integers.
- NumbersPattern = (FloatPattern) | (IntPattern), to match int, float and double.

5.3.8. Method Call

It's used to build the static call regex
L_8={Method Call}

- ClassCall = (\w+\.)+\w+ matches static call of a method or object call method : list.size.
- ClassVariable = \w+\.\w+ , matches class variable like Student.age.
- SimpleArgMethodCall = (ClassVariable) | (NumbersPattern) | \w+ | (LiteralStringPattern), simple argument inside of a method call (): String literal: "Hello world!", Numbers: 32, ClassVariable: City.inhabitant.

- SimpleMethodCall = \s*(ClassCall)\(((SimpleArgMethodCall)(\s*,\s*SimpleArgMethodCall)*)?\s*\)\s*, matches simple method call with no method call as parameter inside the(): list.size(),Student.grade(grade1, grade2, 12.90).
- Inside = (SimpleArgMethodCall) | (Class)\(((SimpleMethodCall)(\s*,\s*SimpleMethodCall)*)?\s*\)\s* | (SimpleMethodCall), inside of a method call is either numbers, variable, class variable, String Literal, nested method call, simple method call.
- MethodCall = \s*(ClassCall)\(((Inside)(\s*,\s*(Inside))*)?\s*\)\s* , matches double nested method call.

5.3.9. Throw Detection

- IC(Import Conflict): some exceptions inside a throw statement needs to be imported to be used.
  Example: throw new IllegalArgumentException("Number must be positive").
- JEA(Java Exception Analysis): metric needs to fetch the exception of a throw statement.
  Example: throw new Exception("ERROR EXCEPTION HAPPENING").

L_9={Throw Statement}

- ThrowPattern = \s*throw\s+new\s+\w+\s*\(\s*(Inside)\s*\)\s*;\s* , matches throw statment :
  throw new IllegalArgumentException("Age must be 18 or older.");

5.3.10. Class

JAX(Java Analyzer): the metric needs to fetch all the classes definition of a java file.

L_{10}={Class Definition}

- AcessModifier = (public\s+ | private\s+ | protected\s+)?, to matches acces modfiers: private, public, protected or none.
- NonAcessModifierClass = (abstract\s+ | final\s+)?, matches abstract, final modifier.
- ModifierClass = (AcessModifier)(NonAcessModifierClass) | (NonAcessModifierClass)(AcessModifier) , matches all possible combination of modifiers : final private , public abstract , abstract ...etc
- ExtendsPattern = (?:\s+extends\s+\w+(\s*<\s*\w+\s*>\s*)?)? , matches extends : extends TreeCell<TreeItemData>.
- ImplementsPattern = (?:\s+implements\s+\w+\s*(\s*,\s*\w+\s*)*)? , macthes one/multiple implements of interfaces : implements Comparable, MathInterface.
- ClassPattern = \s*(ModifierClass)class\s+\w+(TypeParameterGen) (ExtendsPattern) (ImplementsPattern) \s*(?:\{\s*), matches class definition line:
  public class CustomTreeCell extends TreeCell<TreeItemData >{...}

5.3.11. Instanciation

- IC(Import Conflict): to fetch the constructor since some of them needs to be imported to be used.
  Example: try (BufferedReader reader = new BufferedReader(new FileReader(file))).
- SC(Swing Component): to fetch the constructor of swing element.
  Example: mainFrame.setLayout(new BorderLayout());

L_{11}={Line That Contains Instanciation}

- NewPattern = .+ (\(| =)\s*new\s+ .+ , matches line of code that contains instanciation: try(BufferedReader reader = new BufferedReader(new FileReader(file))).

5.3.12. Variable

- IC(Import Conflict): it's needed to fetch the classes from the variables(reference type, type parameter of a collection) because some of them need imports to be used.
  Example: public static ArrayList<ImportStatus >ListImport.
- ER(Encapsulation Rate): the metric needs to obtain the class attribute from the java file to fetch its access modifier.
  Example: private static boolean IsMailUsed = false.
- SC(Swing Component): it's needed to fetch swing element from the variables. Example: JButton button.

L_{12}={Variables}

- PatternAcessModifiers = (private\s+ | protected\s+ | public\s+)?, to matches acces modfiers: private, public, protected or none.
- StaticModifier = (static\s+)?, macthes static modifier.
- FinalModifier = (final\s+)?, matches final modifier.
- VarModifer = (PatternAcessModifiers) (FinalModifier) (StaticModifier) | (PatternAcessModifiers) (StaticModifier) (FinalModifier) | (StaticModifier) (PatternAcessModfiers) (FinalModifier) | (StaticModifier) (FinalModifier) (PatternAcessModifiers) | (FinalModifier) (PatternAcessModifiers) (StaticModifier) | (FinalModifier) (StaticModifier) (PatternAcessModifiers), matches all the possible combination of the modifiers: static final, final private static, public final, etc.
- VariablePattern = (VarModifer) ((?!return\s+)\w+\s+\w+ | (ArrayDeclarationPattern) | (CollectionPattern)\w+)\s*(=\s*.+)?;?, matches Variables: List<Integer>NbList; int a = 0; etc.

5.3.13. Annotation

Import Conflict: needed to fetch the annotation because some of them needs an import to be used.
Example: @FXML.

L_{13}={Annotation Besides Overload and Override}

- AnnotationPatten = \s*@\s*(?!(Overload | Override))\w+ \s*, this matches all anotation beside Override and Overload: @FXML

5.3.14. Static Call

IC(Import Conflict): some static call method and variable class need to be imported to be used.
Example: for (ImportStatus Import: ImportController.ListImport) Encapsulation encapsulation = Encapsulation.EncapsulationFetch(file).

L_{14}={Line That Contains Static Method Call}

- StaticCallPattern= .+ (MethodCall) .+, this matches line of code that contains Static Call: ListImport= ImportStatus.update(file,(ImportStatus.ImportFetch(file)));

6. Conclusion and futur work

The objective of this paper is to strengthen software defect prediction techniques by increasing the set of objectoriented metrics, which leads to decreasing the load, time and effort of software development. In this context, the evaluation of metrics has become crucial to measure the quality, complexity and performance of software. This paper focuses on proposing new object-oriented metrics to better understand contemporary challenges in software development and their contribution to improving development processes by identifying risk areas and guiding code optimization in creating robust and maintainable software. We have proposed six new object-oriented metrics. Similarly, we have proposed a formal model based on regular expressions that allows us to present our metrics under a mathematical model to validate their accuracy. Finally, we conclude with the future perspectives: First, develop a tool based on the proposed medel. Second, working on a prediction model which is based on our proposed metrics as well as some existing ones using artificial intelligence algorithms. Third, empirically study the dependencies between our new metrics and other existing software measurements. Fourth, integrating existing metrics into the software and scaling metrics ie. adapting metrics with other programming paradigms.

References

[1] N. E. Fenton, Software metrics: a practical and rigorous approach, International Thomson Pub., 1996.

[2] V. R. Basili, L. C. Briand, W. L. Melo, A validation of object-oriented design metrics as quality indicators, IEEE Transactions on software engineering 22 (1996) 751–761.

[3] A. Kaur, R. Malhotra, Application of random forest in predicting fault-prone classes, in: 2008 international conference on advanced computer theory and engineering, IEEE, 2008, pp. 37–43.

[4] S. S. Rathore, S. Kumar, An empirical study of some software fault prediction techniques for the number of faults prediction, Soft Computing 21 (2017) 7417–7434.

[5] S. S. Rathore, S. Kumar, A study on software fault prediction techniques, Artificial Intelligence Review 51 (2019) 255–327.

[6] B. Turhan, A. T. Mısırlı, A. Bener, Empirical evaluation of the effects of mixed project data on learning defect predictors, Information and Software Technology 55 (2013) 1101–1118.

[7] Y. Singh, A. Kaur, R. Malhotra, Empirical validation of object-oriented metrics for predicting fault proneness models, Software quality journal 18 (2010) 3–35.

[8] P. Bhattacharya, M. Iliofotou, I. Neamtiu, M. Faloutsos, Graph-based analysis and prediction for software evolution, in: 2012 34th International conference on software engineering (ICSE), IEEE, 2012, pp. 419–429.

[9] D.-L. Miholca, G. Czibula, I. G. Czibula, A novel approach for software defect prediction through hybridizing gradual relational association rules with artificial neural networks, Information Sciences 441 (2018) 152–170.

[10] A. Majd, M. Vahidi-Asl, A. Khalilian, P. Poorsarvi-Tehrani, H. Haghighi, Sldeep: Statement-level software defect prediction using deep-learning model on static code features, Expert Systems with Applications 147 (2020) 113156.

[11] N. A. A. Khleel, K. Nehéz, A novel approach for software defect prediction using cnn and gru based on smote tomek method, Journal of Intelligent Information Systems 60 (2023) 673–707.

[12] Q. Yu, S. Jiang, J. Qian, L. Bo, L. Jiang, G. Zhang, Process metrics for software defect prediction in object-oriented programs, IET Software 14 (2020) 283–292.

[13] A. Tete, F. Toure, M. Badri, Using deep learning and object-oriented metrics to identify critical components in object-oriented systems, in: Proceedings of the 2023 5th World Symposium on Software Engineering, 2023, pp. 48–54.

[14] S. R. Chidamber, C. F. Kemerer, A metrics suite for object oriented design, IEEE Transactions on software engineering 20 (1994) 476–493.

[15] N. I. Churcher, M. J. Shepperd, Towards a conceptual framework for object oriented software metrics, ACM SIGSOFT Software Engineering Notes 20 (1995) 69–75.

[16] F. B. Abreu, R. Carapuça, Object-oriented software engineering: Measuring and controlling the development process, in: Proceedings of the 4th international conference on software quality, volume 186, 1994, pp. 1–8.

[17] F. B. Abreu, M. Goulão, R. Esteves, Toward the design quality evaluation of object-oriented software systems, in: Proceedings of the 5th International Conference on Software Quality, Austin, Texas, USA, 1995, pp. 44–57.

[18] Manju, P. K. Bhatia, A survey of static and dynamic metrics tools for object oriented environment, in: Emerging Research in Computing, Information, Communication and Applications: ERCICA 2020, Volume 2, Springer, 2022, pp. 521–530.

[19] A. Ouellet, M. Badri, Combining object-oriented metrics and centrality measures to predict faults in object-oriented software: An empirical validation, Journal of Software: Evolution and Process 36 (2024) e2548.

[20] M. Sipser, Introduction to the theory of computation, ACM Sigact News 27 (1996) 27–29.

[21] V. Terrón-Macias, J. Mejia, M. A. Canseco-Pérez, M. Muñoz, M. Terrón-Hernández, Fungiregex: A tool for pattern identification in fungal proteomic sequences using regular expressions, Applied Sciences 14 (2024) 4429.

[22] K. Mamouras, A. Chattopadhyay, Efficient matching of regular expressions with lookaround assertions, Proceedings of the ACM on Programming Languages 8 (2024) 2761–2791.