Description of the Full Set of 60 Static Code Features

ACM Reference Format:
. 2025. Description of the Full Set of 60 Static Code Features. In . ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

A CODE COMPLEXITY

These features primarily encompass logical complexity (cyclomatic complexity, cohesion), coupling (dependency relationships, inheritance depth), and theoretical complexity.

Maximum Cyclomatic Complexity (OCmax): This is a logical complexity metric defined in [13], which is the maximum cyclomatic complexity among all non-abstract methods within a class. The base complexity of each method is 1, and the complexity increases by 1 for each occurrence of the following control flow structures: for loops, foreach loops, if statements, do-while loops, conditional expressions (ternary operators), while loops, and each case in a switch statement (excluding consecutive cases). This metric reflects the number of branches in the code and is closely related to the difficulty of understanding a method.

Average Operation Complexity (OCavg): We define OCavg as the average cyclomatic complexity of all non-abstract methods in a class, which is derived from the OCmax feature [13]. This metric reflects the logical complexity of the set of methods in a class on an average basis.

Weighted Methods per Class (WMC): This is a logical complexity metric defined in [6], which is the sum of the complexities of all methods in a class. It is calculated as:

$$\text{WMC} = \sum(\text{cyclomatic complexity of each method})$$

This metric comprehensively reflects the total logical complexity of all methods in a class and is closely related to the overall logical complexity of the method set of the class.

Number of Dependencies (Dcy): This is an object-oriented coupling metric defined in [3], which is the number of other classes that a class depends on. This feature is calculated by constructing a dependency graph combined with graph traversal algorithms and is negatively correlated with the modularity of the system.

Number of Transitive Dependencies (Dcy*): The number of other classes that a class depends on either directly or indirectly, which is derived from the Dcy feature [3]. The difference from Dcy is that it includes both the number of directly dependent classes and the number of indirectly dependent classes. This feature can reflect the deep coupling state of a class in the dependency network.

Number of Dependents (DPT): This is an object-oriented coupling metric defined in [3], which is the number of other classes that directly depend on the current class. We first search for other classes in the project that use the current class, and then count the number of classes that reference the current class as a result.

Number of Transitive Dependents (DPT*): This is an object-oriented coupling feature defined in [3], which is the number of classes that indirectly depend on the current class through a dependency chain. We construct a dependency graph and use graph traversal algorithms to find all transitive dependencies, then count the number of classes that indirectly depend on the current class.

Cyclic Dependencies (Cyclic): This is an object-oriented coupling feature defined in [15], which is the number of classes that each class depends on either directly or indirectly, and these dependencies also depend back on the class either directly or indirectly. It represents the number of other classes involved in cyclic dependencies. For the calculation of this feature, we first construct a dependency graph and collect all direct dependencies. Then we calculate the transitive dependency relationships and identify strongly connected components. Finally, we count the strongly connected components and subtract 1.

Level (Level): This is an object-oriented theoretical complexity feature defined in [11], which measures how many "layers" of classes a class depends on. For a class that does not depend on other classes in the project, its value is 0; for a class with dependencies, its value is equal to the maximum Level value among the classes it depends on plus 1, excluding classes with mutual dependencies or cyclic dependencies. That is:

$$\text{level} = \max(\text{level of dependent classes}) + 1$$

Adjusted Level (Level*): This is an object-oriented theoretical complexity feature defined in [11], which considers the number of classes in cyclic dependencies on the basis of measuring how many "layers" of classes a class depends on. Its calculation method is similar to Level: for a class that does not depend on other classes in the project, its value is 0; for a class with dependencies, its value is equal to the maximum Level* value among the classes it depends on, plus the number of classes that have mutual dependencies or form cyclic dependencies with it. That is:

$$\text{level} = \max(\text{Level* of non-cyclic dependencies}) + \text{number of cyclic dependencies}$$

Package Dependency Count (PDcy): Derived from Robert C. Martin's package-level dependency metric "Efferent Couplings (Ce)" [12], we extend it to the class level. This metric is defined as the number of external packages that a class directly depends on, reflecting the coupling degree between the class and external modules.

Transitive Package Dependency Count (PDcy*): Considering the problem of cyclic dependencies, this feature is introduced on the basis of PDcy [12]. It represents the number of packages that a class directly and indirectly depends on, and this feature is closely related to code structure and modularity.

Package Dependents (PDpt): PDpt is a dependency feature corresponding to PDcy [12], defined as the number of packages that directly or indirectly depend on the current class. It is obtained by

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

counting the number of packages that directly or indirectly depend on the class.

Depth of Inheritance Tree (DIT): This is an object-oriented inheritance depth metric defined in [6], which is the depth of the inheritance tree of each class. DIT is calculated recursively by tracing the inheritance chain of a class from the current class to the root class (java.lang.Object). The depth value increases by 1 for each level up to the parent class. If there is no parent class (i.e., the top of the inheritance chain is reached), 0 is returned. This metric is closely related to the complexity of software design, as an excessively deep inheritance tree makes the code difficult to understand and maintain.

Coupling Between Objects (CBO): This is an object-oriented coupling metric defined in [6], which measures the degree of "coupling" between a class and other classes. Specifically, two classes are coupled when one class depends on the other or is dependent on the other. Its calculation method is as follows:

$$\text{CBO} = \big|\text{Set of classes dependent on the current class} \cup \text{Set of classes the current class depends on}\big|$$

Message Passing Coupling (MPC): Briand proposed a quantification method for message passing coupling, defining MPC as "the number of statements in a class that call methods of other classes" [4]. Based on this definition, we adjust it to: the number of messages (method calls) sent by a class to other classes, excluding calls to methods of the same class. This metric measures the inter-class coupling degree from the perspective of the frequency of dynamic interactions between classes.

Lack of Cohesion of Methods (LCOM): This is an object-oriented cohesion feature defined in [6], which measures the degree of lack of cohesion between methods of a class. We use the LCOM variant designed by Hitz and Montazeri for calculation. First, a relationship graph between methods is constructed, where nodes are methods and edges represent relationships between methods. Then, the number of connected components in this relationship graph is calculated as the result. This feature is closely related to the cohesion and responsibilities of the class.

Response for Class (RFC): Represents the size of the response set of a class, including methods in the class and called external methods [6]. It is obtained by counting all methods defined in the class and all external methods called, and a higher value indicates more complex behaviour of the class.

Number of Queries (Query): The Query metric quantifies the ability of a class to provide information query services, expressed as the number of non-constructor methods with return values in the class (excluding inherited methods) [14]. The formal definition is:

$$\text{Query} = \big|\{m \in M \mid m \text{ is a non-constructor method with a return type} \neq \text{void}\}\big|$$

where M is the set of methods defined in the class. This feature can measure the strength of the information query ability provided by the class and is closely related to the design complexity and responsibility allocation of the code.

Halstead's Bug Prediction (B)

$$B = \frac{V}{\ln(2)} \times \frac{D}{3000}$$

where V is the Halstead Volume and D is the Halstead Difficulty. This metric estimates the number of latent bugs in a program [7].

Halstead Length (N) The total number of operator and operand occurrences [7] in a program:

$$N = N_1 + N_2$$

- N_1: Total number of operator occurrences
- N_2: Total number of operand occurrences

Halstead Vocabulary (n) The total number of distinct operators and operands [7]:

$$n = n_1 + n_2$$

- n_1: Number of distinct operators
- n_2: Number of distinct operands

Halstead Difficulty (D) A measure of program complexity based on operator and operand diversity [7]:

$$D = \frac{n_1}{2} \times \frac{N_2}{n_2}$$

Halstead Volume (V) The information content required to implement the program [7]:

$$V = N \times \log_2(n)$$

Halstead Effort (E) The mental effort required to develop or understand the program [7]:

$$E = D \times V$$

B SEMANTIC ATTRIBUTES

These features primarily focus on comments and documentation information within code to assess its readability and documentation quality.

Comment Lines of Code (CLOC): This feature is an extension of the concept of Source Lines of Code (SLOC) proposed by Nguyen, defined as the total number of lines in a code file that contain comment content [16]. Its calculation rules adhere to strict syntactic parsing, with a formal representation as:

$$\text{CLOC} = |\{l \in L \mid l \text{ contains comment content}\}| \tag{1}$$

where L is the set of code lines after excluding blank lines. CLOC reflects the physical density of comments in the code and is closely related to the code's interpretability.

Comment Ratio (COM_RAT): This is a feature for code comments defined in [5], representing the ratio of the number of comment lines to the total number of code lines (excluding blank lines) in the code. This feature reflects the relative density of comments in the code and is a classic feature for assessing code readability. Its formal representation is:

$$\text{COM\_RAT} = \frac{\text{CLOC}}{|L|} \tag{2}$$

where $|L|$ denotes the cardinality of the set L of non-blank code lines.

Javadoc Method Coverage (JM): JM is closely related to the completeness of method-level documentation. JM is defined as:

$$\text{JM} = \frac{|\text{methods with Javadoc comments}|}{|\text{total methods in the class}|} \times 100\%. \quad (3)$$

Javadoc Field Coverage (JF): This is a feature for code comments defined in [17], quantifies the documentation level of fields within a class. It is defined as:

$$\text{JF} = \frac{|\text{fields with Javadoc comments}|}{|\text{total fields in the class}|} \times 100\%. \quad (4)$$

Javadoc Lines of Code (JLOC): JLOC is a refined metric of CLOC, specifically counting the number of lines of comments that conform to the Javadoc specification [17]. This metric measures the density of comments in the code that follow standard API documentation conventions and is strongly associated with the completeness of API documentation. Its calculation is:

$$\text{JLOC} = |\{l \in L \mid l \text{ belongs to a Javadoc comment block}\}| \quad (5)$$

True Comment Ratio (TCOM_RAT): Defined as the ratio of the number of true comment lines to the total number of code lines (excluding blank lines). Here, true comment lines refer to those that exclude automatically generated comments (e.g., auto-generated TODO comments) and only include meaningful documentation comments or explanatory comments. This metric reflects the quality density of effective comments in the code and can be used to assess code readability and documentation completeness.

Number of TODO Comments (TODO): This refers to the total number of TODO comments within a class [18].

C PROGRAM SCALE

These features are primarily used to quantify the physical size of code (e.g., lines of code, number of methods/attributes) and the scale of logical structures (e.g., parameters, inner classes).

Lines of Code (LOC): This is a metric defined in [5], which is the total number of lines of code in a class. As a fundamental indicator for measuring code size, it includes comment lines but excludes pure blank lines (lines containing only spaces, tabs, etc., are not counted).

Non-Comment Lines of Code (NCLOC): NCLOC is a derivative indicator of LOC [5], defined as:

$$\text{NCLOC} = \text{LOC} - |\{l \in L \mid l \text{ is a pure comment line}\}|$$

By eliminating the interference of comments, this indicator accurately reflects the scale of actual executable code.

Class Size Attributes (CSA): CSA is defined as the number of non-static attributes defined in a class [5]. Its formal representation is:

$$\text{CSA} = \left|\left\{f \in F \mid f \text{ is a non-static field defined in the current class}\right\}\right|$$

where F is the set of all fields in the class.

Class Size Operations (CSO): As a dual indicator of CSA, CSO is defined as the number of non-static methods defined in a class [5]. Its formal representation is:

$$\text{CSO} = \left|\left\{m \in M \mid m \text{ is a non-static method defined in the current class}\right\}\right|$$

where M is the set of all methods in the class.

Class Size (Operations + Attributes) (CSOA): This is a metric for measuring the logical scale of a class, which is the total number of operations and attributes in the class, i.e., the sum of CSA and CSO.

Maximum Operation Size (OSmax): This is a metric for measuring the structural scale of a class defined in [10], referring to the number of statements in the largest method within the class. Its formal representation is:

$$\text{OSmax} = \max(\{\text{number of statements in method } m \mid m \in M\})$$

Average Operation Size (OSavg): This is a derivative indicator of OPavg [9], defined as the average number of statements per method in the class. It is calculated as:

$$\text{OSavg} = \frac{\sum_{m \in M} (\text{number of statements in method } m)}{|M|}$$

Average Number of Parameters (OPavg): This is a metric for measuring the structural scale of a class defined in [9], referring to the average number of parameters per method in the class. Here, we use OPavg instead of NPAVG in the citation, and this indicator is closely related to the complexity of method interfaces. It is calculated as:

$$\text{OPavg} = \frac{\sum_{m \in M} (\text{number of parameters in method } m)}{|M|}$$

Number of Operations Added (NOAC): This is a metric for measuring the structural scale of a class defined in [9], referring to the number of new methods added by a subclass. Its formal representation is:

$$\text{NOAC} = \left|\left\{m \in M_{\text{sub}} \mid m \notin M_{\text{super}} \text{ and } m \text{ is not an overridden method}\right\}\right|$$

where M_{sub} is the set of methods in the subclass, and M_{super} is the set of methods in the parent class. Here, we use NOAC instead of NMA in the citation; this indicator only counts newly defined methods in the class, excluding methods inherited from the parent class and overridden methods.

Number of Attributes Added (NAAC): As a dual indicator of NOAC [9], NAAC is defined as the number of new attributes added in the class. Specifically, this indicator first counts the attributes directly defined in the class, then excludes attributes inherited from the parent class, and finally calculates the number of newly added fields in the current class as the result.

Number of Operations Overridden (NOOC): This is a metric for measuring the logical scale of a class, referring to the number of methods in a subclass that override methods in the parent class. Here, we use NOOC instead of NMO in [9]; this indicator first checks each method in the class, then determines whether it uses the @Override annotation or overrides a parent class method, and finally counts the number of overridden methods as the result.

Number of Operations Inherited (NOIC): This is a derivative indicator of NOAC [9], defined as the total number of operations (or methods) inherited by a class. Specifically, it first traverses all methods of the class, then excludes constructors, private methods, static methods, abstract methods, and methods defined by the current class itself; the remaining number of methods is the value of NOIC.

Number of Children (NOC): This is an object-oriented metric for measuring the structural scale of a class defined in [6], referring to the total number of direct subclasses of a class. This indicator can also help evaluate the inheritance hierarchy and reuse of classes. A high NOC value indicates that the class is widely inherited, suggesting it may be a well-designed base class/superclass.

Number of Constructors (CONS): Represents the total number of constructors declared in a class [10]. This metric only counts explicitly declared constructors in the class; if no constructors are declared (even if a default constructor exists), the value of the indicator is 0.

Number of Implemented Interfaces (INNER): Considering the need to characterize the scale of a class from the perspective of interface implementation, we define this indicator as the total number of interfaces implemented by the class [8]. Specifically, this feature counts all distinct interfaces implemented directly or indirectly by a class, including interfaces implemented directly and interface implementations inherited from parent classes.

Number of Inner Classes (Inner): This is a derivative indicator of NOAC [9], defined as the number of inner classes or interfaces contained in a class.

Number of Type Parameters (NTP): This is a metric for measuring the structural scale of a class defined in [2], referring to the number of type parameters in the class. A high NTP value indicates that the class uses more type parameters, which may lead to an increase in class scale.

Number of Statements (STAT): Defined as the number of valid statements in a method. It first traverses all PSI statement nodes, then excludes empty statements and block statements, and finally counts the actual executable statements as the result.

D USAGE SCENE

Drawing upon developers' intuition and empirical studies, we have identified 10 typical code scenarios, with a primary focus on code functionality and responsibilities.

String Processing: This scenario encompasses the creation, manipulation, parsing, formatting, or conversion of strings [1]. Examples include processing text data, generating reports, sanitising inputs, and methods such as string splitting, concatenation, and regular expression matching.

File Operations: This scenario involves interactions with the file system and manipulation of file contents, specifically including file creation, reading, and the parsing and management of file paths. Examples include reading configuration files and exporting data to CSV.

Database Operations: This scenario primarily handles database connections, queries, updates, or transaction management. Examples include executing SQL queries, manipulating ORMs (Object-Relational Mappers), managing data storage, or defining data models.

Mathematical Computation: This scenario primarily performs numerical computations, algorithm implementations, or statistical analyses, encompassing mathematical functions, statistical models, or optimisation algorithms. Examples include handling financial calculations, machine learning predictions, or geometric operations.

User Interface: This scenario manages user interactions, interface rendering, or event handling. Examples include building web frontends, desktop GUIs, or mobile interfaces.

Business Logic: This scenario primarily implements specific business rules or domain logic, which are specific to the application context. Examples include user permission validation, workflow engines, etc.

Data Structures and Algorithms: This scenario manages data collections and implements efficient storage or manipulation. Examples include handling lists, trees, graphs, traversing data structures, or optimising methods (e.g., sorting, searching).

System and Tools: This scenario primarily provides general auxiliary functionalities or system-level operations, such as logging, error handling, or process management. Other categories typically reuse code in this category.

Concurrency and Multithreading: This scenario primarily deals with parallel execution, thread management, or resource sharing. Examples include implementing asynchronous tasks or avoiding race conditions.

Exception Handling: This scenario is responsible for uniformly managing the system's exception handling mechanisms, including the capture, classification, handling, and reporting of exceptions, as well as the implementation of exception recovery strategies, etc.

REFERENCES

[1] Azat Abdullin, Pouria Derakhshanfar, and Annibale Panichella. 2025. Test Wars: A Comparative Study of SBST, Symbolic Execution, and LLM-Based Approaches to Unit Test Generation. In *ICST*. IEEE, 221–232.

[2] Joshua Bloch. 2017. *Effective java*. Addison-Wesley Professional.

[3] Lionel C. Briand, John W. Daly, and Jürgen Wüst. 1999. A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Trans. Software Eng.* 25, 1 (1999), 91–121.

[4] Lionel C. Briand, John W. Daly, and Jurgen K Wust. 2002. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on software Engineering* 25, 1 (2002), 91–121.

[5] Sonal Chawla and Gagandeep Kaur. 2013. Comparative Study of the Software Metrics for the complexity and Maintainability of Software Development. *International Journal of Advanced Computer Science and Applications* 4, 9 (2013).

[6] Shyam R Chidamber and Chris F Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on software engineering* 20, 6 (1994), 476–493.

[7] Bill Curtis, Sylvia B. Sheppard, Phil Milliman, MA Borst, and Tom Love. 1979. Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. *IEEE Transactions on software engineering* 2 (1979), 96–104.

[8] Andrea Dobberfuhl and Mark W Lange. 2009. Interfaces per module: Is there an ideal number?. In *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, Vol. 48999. 1373–1385.

[9] Khaled El Emam, Saïda Benlarbi, Nishith Goel, and Shesh N. Rai. 2001. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering* 27, 7 (2001), 630–650.

[10] Martin Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[11] John Lakos. 1996. Large-scale C++ software design. *Reading, MA* 173 (1996), 217–271.

[12] Robert Martin. 1994. OO design quality metrics. *An analysis of dependencies* 12, 1 (1994), 151–170.

[13] Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering* 4 (1976), 308–320.

[14] Bertrand Meyer. 1997. *Object-oriented software construction*. Vol. 2. Prentice hall Englewood Cliffs.

[15] Tosin Daniel Oyetoyan, Daniela S. Cruzes, and Reidar Conradi. 2013. A study of cyclic dependencies on defect profile of software components. *J. Syst. Softw.* 86, 12 (2013), 3162–3182.

[16] Robert E Park. 1992. *Software size measurement: A framework for counting source statements*. Technical Report.

[17] Pooja Rani, Arianna Blasi, Nataliia Stulova, Sebastiano Panichella, Alessandra Gorla, and Oscar Nierstrasz. 2023. A decade of code comment quality assessment: A systematic literature review. *J. Syst. Softw.* 195 (2023), 111515.

[18] Margaret-Anne Storey, Jody Ryall, R Ian Bull, Del Myers, and Janice Singer. 2008. Todo or to bug: Exploring how task annotations play a role in the work practices of software developers. In *Proceedings of the 30th international conference on Software engineering*. 251–260.