Code Metrics :

1. Class Lines of code (CLOC)
2. Cyclomatic Complexity
3. Weighted Method Count (WMC)
4. Coupling Between the Objects (CBO)
5. Lack of Cohesion of Methods (LCOM)
6. Depth of Inheritance tree (DIT)

Tools :

1. CodeMR
2. PMD

1. CLASS LINES OF CODE (CLOC)

- **Type :** Size Metric
- CLOC measures the number of executable lines within a class.
- Observed Value : 754
- Location : weblogger/business/jpa/JPAWeblogEntryManagerImpl
- **Implications**
    - Software Quality
        - High CLOC values indicate oversized classes which are difficult to understand and reduces the overall code readability and increases the likelihood of defects.
    - Maintainability
        - Large classes are harder to modify and extend because changes tend to affect multiple responsibilities within same class
        - This increases the maintenance effort and risk of errors.
    - Potential Performance Issues
        - CLOC does not directly impact runtime performance, but larger classes often contain inefficient or redundant logic, which may indirectly contribute to performance overhead.
- **Refactoring Guidance**
    - Classes with high CLOC can be refactored through techniques like **Extract class or Modularization** where the responsibilities are redistributed into smaller classes.
- **Reflection on Current state**
    - High CLOC values reflect the project's incremental growth over time and suggest the presence of legacy classes that would benefit from modularization.

2. CYCLOMATIC COMPLEXITY (CC)

- **Type :** Complexity Metric
- Measures the number of independent execution paths within a method based on control flow constructs based on conditional statements and loops.
- Observed Value : High
- Location
    - weblogger/util/Utilities
    - weblogger/business/jpa/jpaMediaFileManagerImpl
    - weblogger/ui/struts2/editor/Comments
    - weblogger/ui/struts2/editor/EntryBean
    - weblogger/pojos/wrapper/WeblogEntryWrapper
- **Implications**
    - Software Quality
        - High cyclomatic complexity reduces code clarity and increases the probability of logical errors as the number of execution paths becomes harder to manage and verify.
    - Maintainability
        - High complex methods are difficult to test and debug.
        - Even small changes require extensive retesting making the maintenance costly and error prone.
    - Potential Performance Issues
        - Complex control flows may result in inefficient branching and repeated evaluations, impacting execution time.
- **Refactoring Guidance**
    - High complexity methods should be simplified by breaking them into smaller methods, reducing the nesting conditionals or applying polymorphism to replace the complex decision logic.
- **Reflection on Current state**
    - The complexity reflects the platform's rich feature set but also highlights areas where logic has become overly dense and could be simplified.

3. WEIGHTED METHOD COUNT (WMC)

- **Type :** OOP - Complexity
- Represents the sum of complexities of all methods within a class.
- Observed Value : 188 (Very high)

- Location : weblogger/business/jpa/JPAWeblogEntryManagerImpl
- **Implications**
    - Software Quality
        - High WMC suggest that a class contains highly complex methods indicating poor design and violation of object oriented principles.
    - Maintainability
        - Classes with high WMC are harder to understand and modify as there are multiple complex behaviors within a single class.
        - This increases the coupling changes and functionality
    - Potential Performance Issues
        - Classes with many complex methods may introduce redundant computations which will negatively affect the performance when such classes are more widely used.
- **Refactoring Guidance**
    - Classes with high WMC should be restructured by splitting responsibilities across multiple classes improving the modularity and adherence to object oriented principles.
- **Reflection on Current state**
    - High WMC reflects accumulated responsibilities in core service classes, common in long-lived open-source projects.

4. COUPLING BETWEEN THE OBJECTS (CBO)

- **Type :** OOP - Coupling
- Measures the number of other classes to which a given class is coupled.
- Observed Value : 33
- Location : weblogger/ui/struts2/editor/EntryEdit
- **Implications**
    - Software Quality
        - High Coupling reduces modularity and increases the dependency among classes making the system more fragile and susceptible to defects when changes occur.
    - Maintainability
        - Strong inter class dependencies mean that the modifications in one class can require changes in several

        - others which increases the maintenance effort and reduces the flexibility.
    - Potential Performance Issues
        - Excessive coupling can lead to frequent object creation, method callings and increased communication overhead between classes which degrades the runtime performance.
- **Refactoring Guidance**
    - High CBO values suggest the need for dependency reduction, which can be done by the use of interfaces.
- **Reflection on Current State**
    - The observed coupling reflects tight integration between platform components, suggesting the need for improved abstraction and dependency management.

5. LACK OF COHESION OF METHODS

- **Type:** OOP - Cohesion
- Measures how closely related the methods of a class are based on shared instance variables.
- Observed Value : High
- Location :
    - weblogger/business/jpa/JPAWeblogEntryManagerImpl
    - weblogger/pojos/WeblogEntry
    - weblogger/pojos/Weblog
    - weblogger/business/jpa/JPAWeblogmanagerImpl
- **Implications**
    - Software Quality
        - High LCOM values indicate that methods within a class are weakly related, reflecting poor design and reduced conceptual integrity of classes.
    - Maintainability
        - Low cohesion makes classes difficult to maintain because developers must deal with unrelated functionalities grouped together, increasing the risk of un-intended side effects during modifications.
    - Potential Performance Issues
        - Low cohesion may cause unnecessary data sharing and repeated access to unrelated fields leading to inefficient memory usage and potential performance degradation.
- **Refactoring Guidance**

    - Classes with high LCOM should split into smaller, more cohesive classes each focusing on single responsibility.
- **Reflection on Current State**
    - This reflects design erosion over time, where classes have absorbed additional responsibilities instead of being decomposed.

6. Depth of Inheritance Tree (DIT)

- **Type:** OOP - Inheritance
- Measures the number of levels a class is removed from the root of the inheritance hierarchy.
- Observed Value : 2
- Location : weblogger/ui/struts2/editor/Comments
- **Implications**
    - Software Quality
        - Deep Inheritance hierarchy makes program behavior harder to understand as functionality is inherited from multiple ancestor classes.
    - Maintainability
        - High DIT values increase the maintenance complexity since the changes in base classes can propagate unpredictably to subclasses.
    - Potential Performance Issues
        - Deep inheritance hierarchy may introduce overhead due to dynamic method dispatch and increased method resolution time which can affect performance in inheritance heavy systems.
- **Refactoring Guidance**
    - High DIT values may indicate overuse of inheritance.
    - In such cases composition should be preferred over inheritance or inheritance hierarchy should be flattened where possible.
- **Reflection on Current State**
    - DIT values indicate structured reuse but also highlight areas where inheritance may be overused instead of composition.