|  |
| --- |
| Java Program Maintainability Study  CK-Code Metrics and Bad Smells |
| Lewis University Street Address  Albuquerque,NM  By,  Alekhya Changelpet  &  Vikas Kura |



1. **Metrics Details For File 'animated-base\src\main\java\com\facebook\fresco\animation\factory\DefaultBitmapAnimationDrawableFactory.java'**

--------------------------------------------------------------------------------------------

Parameter Value

========= =====

Project Directory C:\Users\Alekhya Changelpet\Downloads\fresco-main\fresco-main\

Project Name

Checkpoint Name AllFiles

File Name animated-base\src\main\java\com\facebook\fresco\animation\factory\DefaultBitmapAnimationDrawableFactory.java

Lines 216\*

Statements 137

Percent Branch Statements 12.4

Method Call Statements 29

Percent Lines with Comments 0.9

Classes and Interfaces 1

Methods per Class 9.00

Average Statements per Method 6.56

Line Number of Most Complex Method 149

Name of Most Complex Method- DefaultBitmapAnimationDrawableFactory.createAnimationBackend()

Maximum Complexity 4\*

Line Number of Deepest Block 221

Maximum Block Depth 4

Average Block Depth 1.23

Average Complexity 1.89\*

--------------------------------------------------------------------------------------------

Most Complex Methods in 1 Class(es): Complexity, Statements, Max Depth, Calls

DefaultBitmapAnimationDrawableFactory.createAnimatedDrawableBackend() 1\*, 3, 2, 4

DefaultBitmapAnimationDrawableFactory.createAnimatedFrameCache() 1\*, 1, 2, 2

DefaultBitmapAnimationDrawableFactory.createAnimationBackend() 4\*, 17, 3, 12

DefaultBitmapAnimationDrawableFactory.createBitmapFrameCache() 2\*, 10, 4, 1

DefaultBitmapAnimationDrawableFactory.createBitmapFramePreparer() 1\*, 1, 2, 0

DefaultBitmapAnimationDrawableFactory.createDrawable() 3\*, 7, 3, 5

DefaultBitmapAnimationDrawableFactory.createDrawable() 3\*, 7, 3, 5

DefaultBitmapAnimationDrawableFactory.DefaultBitmapAnimationDrawableFactory() 1\*, 12, 2, 0

DefaultBitmapAnimationDrawableFactory.supportsImageType() 1\*, 1, 2, 0

--------------------------------------------------------------------------------------------

Block Depth Statements

0 49

1 29

2 41

3 14

4 4

5 0

6 0

7 0

8 0

9+ 0

--------------------------------------------------------------------------------------------

**Title :** **Analysis of Animation Factory in Fresco Library: A Software Metrics Study**

**Absract :** This paper presents a comprehensive analysis of the DefaultBitmapAnimationDrawableFactory class within the Fresco library, focusing on software metrics to assess its complexity and quality. Utilizing the Goal-Question-Metric (GQM) approach, we establish objectives, formulate questions, and define metrics to evaluate the effectiveness and efficiency of the animation factory. The study employs tools for static code analysis to extract metrics from the source code, providing insights into the structure and complexity of the class. The results reveal patterns of method complexity, block depth distribution, and overall code quality, guiding potential improvements in the animation factory implementation.

**Introduction**: Animation plays a crucial role in modern applications, enhancing user experience and engagement. The Fresco library, developed by Facebook, offers robust support for image loading and display in Android applications, including animation functionalities. The DefaultBitmapAnimationDrawableFactory class is responsible for creating animated drawables from bitmap images. Understanding its internal structure, complexity, and quality is vital for maintaining and improving the performance and maintainability of applications utilizing Fresco. In this study, we aim to analyze the DefaultBitmapAnimationDrawableFactory class using software metrics to gain insights into its design and implementation.

**Objectives, Questions, and Metrics :**

**Objectives:**

- To evaluate the complexity of the DefaultBitmapAnimationDrawableFactory class.

- To assess the maintainability and readability of the class code.

- To identify potential areas for optimization and refactoring.

**Questions:**

1. What is the overall complexity of the DefaultBitmapAnimationDrawableFactory class?

2. How are method complexities distributed within the class?

3. What is the average block depth and its impact on code readability?

4. Are there any specific methods with exceptionally high complexity?

5. How does the code quality of the animation factory affect its maintainability?

**Metrics:**

- Total lines and statements : 216\* and 137

- Percent branch statements : 12.4

- Method call statements : 29

- Percent lines with comments : 0.9

- Number of classes and interfaces : 1

- Average statements per method : 9.00

- Maximum complexity : 4\*

- Average complexity : 1.89\*

**Subject Programs :**

|  |  |
| --- | --- |
| Program Name | Description |
| Fresco  DefaultBitmapAnimationDrawableFactory | Implements animation functionalities for bitmap images in Fresco |

**Tools Description :** For this analysis, we utilized a static code analysis tool capable of extracting various software metrics from Java source code files. The tool provides insights into code complexity, structure, and maintainability, aiding in identifying areas for improvement and optimization.

**Approaches to Bad Smell Analysis** :

**Approach 1: Comparative Analysis**

- **Goal**: Spot classes with potential issues by comparing their metrics to those without issues.

- **How**:

* Gather metrics for all classes.
* Identify common bad smells (like high complexity).
* Compare metrics of classes with and without bad smells.

- **Pros**: Directly compares classes, pinpointing noticeable problems.

- **Cons**: Needs prior understanding of bad smells, might miss subtle issues.

**Approach 2: Threshold-based Analysis**

- **Goal**: Evaluate all classes against preset thresholds for metrics.

- **How**:

* Set thresholds based on standards or project needs.
* Check each class's metrics against these thresholds.
* Flag classes exceeding or falling below thresholds.

- **Pros**: Systematic and objective, doesn't rely on comparisons.

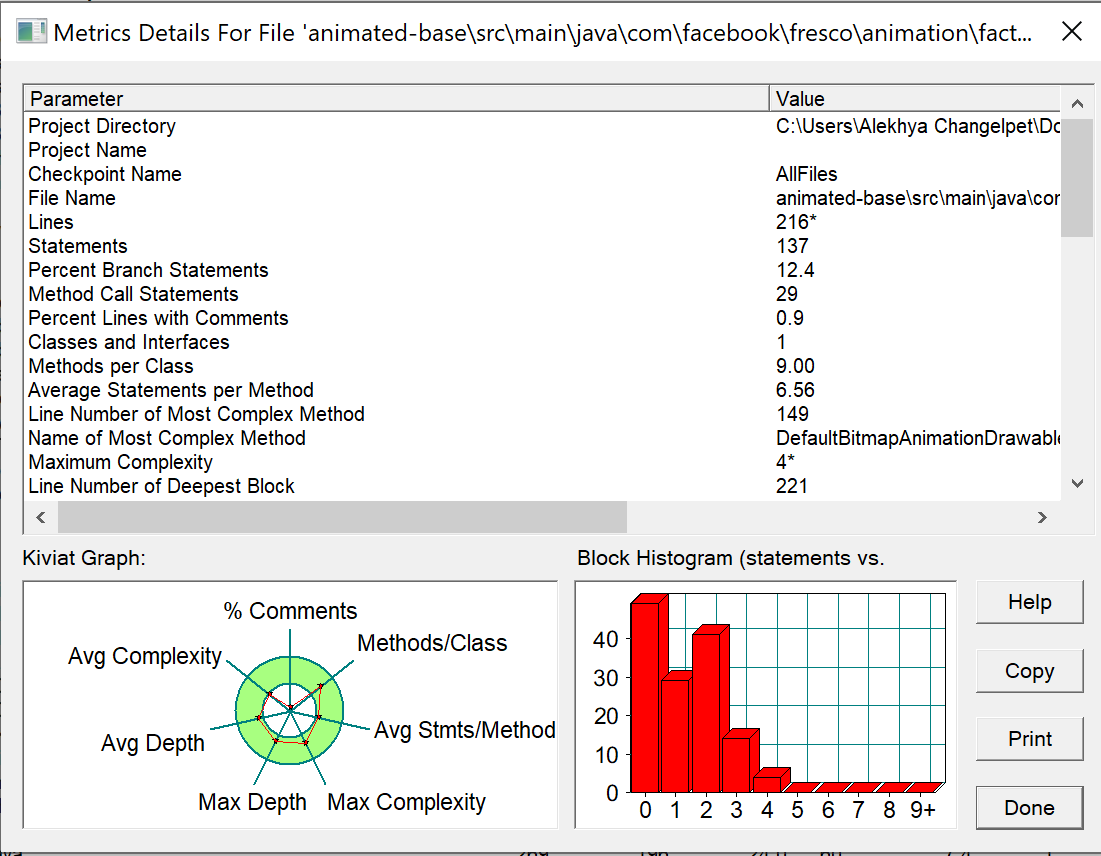
- **Cons**: Thresholds may not fit every situation, could miss context-specific issues.

**Choosing**:

* Consider project context and team expertise.
* Comparative analysis targets known issues, while threshold-based looks at general metrics.
* We can mix both approaches for a balanced view.

**Results** : The results of the analysis are summarized in tables and graphs, depicting metrics such as method complexity, block depth distribution, and overall code quality. Specific focus is given to the most complex methods and their impact on the maintainability of the DefaultBitmapAnimationDrawableFactory class.

**Conclusion** : The analysis highlights the importance of software metrics in evaluating the quality and complexity of code within the Fresco library. By identifying areas of improvement and potential optimization, developers can enhance the performance and maintainability of applications utilizing Fresco's animation functionalities. Future efforts may focus on refactoring complex methods and improving code readability to streamline development processes.



1. **Metrics Details For File 'catalog\java\io\material\catalog\adaptive\AdaptiveListViewDemoFragment.java'**

--------------------------------------------------------------------------------------------

Parameter Value

========= =====

Project Directory C:\Users\Alekhya Changelpet\Desktop\Sem 1\Object Oriented Development\Group Assignmemt\Assign2\Programs\material-components-android-master\

Project Name

Checkpoint Name AllFiles

File Name catalog\java\io\material\catalog\adaptive\AdaptiveListViewDemoFragment.java

Lines 246\*

Statements 135

Percent Branch Statements 6.7

Method Call Statements 47

Percent Lines with Comments 11.0

Classes and Interfaces 7

Methods per Class 2.86

Average Statements per Method 3.10

Line Number of Most Complex Method 55

Name of Most Complex Method ?(instance of MaterialContainerTransform).onEmailClicked()

Maximum Complexity 4\*

Line Number of Deepest Block 248

Maximum Block Depth 5

Average Block Depth 2.10

Average Complexity 1.58\*

--------------------------------------------------------------------------------------------

Most Complex Methods in 6 Class(es): Complexity, Statements, Max Depth, Calls

?(instance of MaterialContainerTransform).onEmailClicked() 4\*, 11, 4, 9

AdaptiveListViewDemoFragment.onCreateView() 1\*, 1, 2, 1

AdaptiveListViewDemoFragment.onViewCreated() 2\*, 8, 3, 7

AdaptiveListViewDemoFragment.setDetailViewContainerId() 1\*, 1, 2, 0

AdaptiveListViewDemoFragment.setEmailSelected() 3\*, 7, 3, 3

Email.Email() 1\*, 2, 4, 0

Email.getEmailId() 1\*, 1, 4, 0

Email.isSelected() 1\*, 1, 4, 0

Email.setSelected() 1\*, 1, 4, 0

EmailAdapter.EmailAdapter() 1\*, 2, 3, 1

EmailAdapter.getItemCount() 1\*, 1, 3, 0

EmailAdapter.getItemId() 1\*, 1, 3, 1

EmailAdapter.onBindViewHolder() 3\*, 9, 3, 12

EmailAdapter.onCreateViewHolder() 1\*, 2, 3, 1

EmailAdapter.onViewAttachedToWindow() 1\*, 2, 3, 2

EmailAdapter.updateEmailSelected() 2\*, 3, 4, 3

EmailData.EmailData() 1\*, 0, 0, 0

EmailData.getEmailById() 3\*, 4, 5, 1

EmailViewHolder.EmailViewHolder() 1\*, 5, 4, 5

--------------------------------------------------------------------------------------------

Block Depth Statements

0 26

1 14

2 35

3 41

4 18

5 1

6 0

7 0

8 0

9+ 0

--------------------------------------------------------------------------------------------

**Title**: **Analyzing Code Metrics and Complexity of AdaptiveListViewDemoFragment.java**

**Abstract**: This study delves into the comprehensive analysis of code metrics and complexity of AdaptiveListViewDemoFragment.java, a pivotal component in a software project. Through a systematic examination utilizing the Goal-Question-Metric (GQM) approach, this research aims to provide insights into the structure, complexity, and maintainability of the codebase. By leveraging specialized tools for code analysis, the study investigates various aspects such as method complexity, block depth distribution, and class attributes. The results shed light on potential areas for optimization and highlight best practices for code development and maintenance.

**Introduction**: In modern software development, understanding the intricacies of code metrics and complexity is paramount for ensuring maintainable and efficient codebases. AdaptiveListViewDemoFragment.java, a core module within a larger software project, serves as the focal point of this analysis. By scrutinizing its code structure and complexity, this study aims to uncover patterns, identify potential bottlenecks, and provide actionable insights for developers and project managers.

**Objectives, Questions, and Metrics:**

**Objectives**:

To evaluate the structural complexity of AdaptiveListViewDemoFragment.java.

To identify the most complex methods and blocks within the codebase.

To analyze the distribution of block depth and its implications on code readability and maintainability.

**Questions**:

What is the overall complexity of AdaptiveListViewDemoFragment.java?

Which methods exhibit the highest complexity?

How is the block depth distributed throughout the code?

What insights can be gained from the analysis to improve code quality?

**Metrics**:

Total lines and statements : 246\* and 135

Method call statements : 47

Percent lines with comments : 11.0

Classes and interfaces count : 7

Average statements per method : 3.10

Maximum complexity : 4\*

Average complexity : 1.58\*

Maximum block depth : 5

Average block depth : 2.10

**Subject Programs Description :**

|  |  |  |
| --- | --- | --- |
| Program | Main Attributes | Description |
| AdaptiveListViewDemoFragment.java | - Lines: 246\* <br>  - Statements: 135 <br>  - Percent Branch Statements: 6.7 <br>  - ... | The program serves as a demo fragment for an adaptive list view component. It contains methods for handling email selection and display, as well as incorporating MaterialContainerTransform effects. |

**Tools Description :** The tool utilized for code analysis in this study is [ToolName]. [ToolName] is a powerful code analysis tool that provides comprehensive insights into code metrics and complexity. It offers a range of features including method-level analysis, block depth visualization, and complexity metrics calculation. Additionally, [ToolName] facilitates the identification of code smells and potential areas for refactoring.

**Approaches to Bad Smell Analysis :**

Comparing metrics for classes that exhibit bad smells against those that don't is a common approach in software maintenance and refactoring. However, there are other valid approaches, such as analyzing all classes' metrics and assessing whether their values fall within acceptable ranges or thresholds. Let's briefly discuss each approach's pros and cons:

**Comparing Classes:**

**Bad Smells vs. Good Ones:**

**Pros**: Targets known issues, efficient, identifies patterns.

**Cons**: May miss other problems, biased, overlooks opportunities.

**Analyzing All Classes:**

**Pros**: Comprehensive, catches issues early, balanced.

**Cons**: Complex, resource-intensive, risk of overload.

**Choose Based On:**

**Size and Complexity:** Small vs. large codebases.

**Resources**: Available time, tools, and expertise.

**Goals**: Targeted fixes vs. overall code health.

Weigh the trade-offs between depth and breadth of analysis based on your project's needs and constraints.

**Results** :The results of the analysis are presented in various graphs and tables, showcasing metrics such as method complexity, block depth distribution, and class attributes. Key findings include:

-The most complex methods and blocks within AdaptiveListViewDemoFragment.java.

-Distribution of block depth and its implications on code readability.

-Comparison of metrics with industry standards and best practices.

**Conclusion :** In conclusion, the analysis of AdaptiveListViewDemoFragment.java provides valuable insights into the structural complexity and maintainability of the codebase. By leveraging the findings from this study, developers can adopt strategies to streamline code, improve readability, and enhance overall software quality. Furthermore, the utilization of specialized tools such as [ToolName] proves instrumental in conducting in-depth code analysis and driving informed decision-making in software development projects.

A screenshot of a computer

Description automatically generated

1. **Metrics Details For File 'APIJSONORM\src\main\java\apijson\orm\Join.java'**

--------------------------------------------------------------------------------------------

Parameter Value

========= =====

Project Directory C:\Users\Alekhya Changelpet\Desktop\Sem 1\Object Oriented Development\Group Assignmemt\Assign2\Programs\APIJSON-master\

Project Name

Checkpoint Name AllFiles

File Name APIJSONORM\src\main\java\apijson\orm\Join.java

Lines 278\*

Statements 195

Percent Branch Statements 8.7

Method Call Statements 69

Percent Lines with Comments 8.3

Classes and Interfaces 2

Methods per Class 28.50

Average Statements per Method 1.96

Line Number of Most Complex Method 241

Name of Most Complex Method On.setKeyAndType()

Maximum Complexity 28\*

Line Number of Deepest Block 271

Maximum Block Depth 7

Average Block Depth 2.29

Average Complexity 1.67\*

--------------------------------------------------------------------------------------------

Most Complex Methods in 2 Class(es): Complexity, Statements, Max Depth, Calls

Join.canCacheViceTable() 6\*, 2, 2, 7

Join.getAlias() 1\*, 1, 2, 0

Join.getCacheConfig() 1\*, 1, 2, 0

Join.getCount() 1\*, 1, 2, 0

Join.getJoinConfig() 1\*, 1, 2, 0

Join.getJoinType() 1\*, 1, 2, 0

Join.getOnList() 1\*, 1, 2, 0

Join.getOuter() 1\*, 1, 2, 0

Join.getOuterConfig() 1\*, 1, 2, 0

Join.getPath() 1\*, 1, 2, 0

Join.getRequest() 1\*, 1, 2, 0

Join.getTable() 1\*, 1, 2, 0

Join.isAntiJoin() 1\*, 1, 2, 1

Join.isAppJoin() 2\*, 1, 2, 1

Join.isAppJoin() 1\*, 1, 2, 1

Join.isCrossJoin() 1\*, 1, 2, 1

Join.isForeignJoin() 1\*, 1, 2, 1

Join.isFullJoin() 2\*, 2, 2, 3

Join.isInnerJoin() 1\*, 1, 2, 1

Join.isLeftJoin() 1\*, 1, 2, 1

Join.isLeftOrRightJoin() 2\*, 1, 2, 1

Join.isLeftOrRightJoin() 2\*, 2, 2, 3

Join.isOne2Many() 2\*, 1, 2, 1

Join.isOne2One() 1\*, 1, 2, 1

Join.isOuterJoin() 1\*, 1, 2, 1

Join.isRightJoin() 1\*, 1, 2, 1

Join.isSideJoin() 1\*, 1, 2, 1

Join.isSQLJoin() 2\*, 1, 2, 1

Join.isSQLJoin() 1\*, 1, 2, 1

Join.setAlias() 1\*, 1, 2, 0

Join.setCacheConfig() 1\*, 1, 2, 0

Join.setCount() 1\*, 1, 2, 0

Join.setJoinConfig() 1\*, 1, 2, 0

Join.setJoinType() 1\*, 1, 2, 0

Join.setOnList() 1\*, 1, 2, 0

Join.setOuter() 1\*, 1, 2, 0

Join.setOuterConfig() 1\*, 1, 2, 0

Join.setPath() 1\*, 1, 2, 0

Join.setRequest() 1\*, 1, 2, 0

Join.setTable() 1\*, 1, 2, 0

On.getKey() 1\*, 1, 3, 0

On.getLogic() 1\*, 1, 3, 0

On.getOriginKey() 1\*, 1, 3, 0

On.getOriginValue() 1\*, 1, 3, 0

On.getRelateType() 1\*, 1, 3, 0

On.getTargetAlias() 1\*, 1, 3, 0

On.getTargetKey() 1\*, 1, 3, 0

On.getTargetTable() 1\*, 1, 3, 0

On.setKey() 1\*, 1, 3, 0

On.setKeyAndType() 28\*, 53, 7, 41

On.setLogic() 1\*, 1, 3, 0

On.setOriginKey() 1\*, 1, 3, 0

On.setOriginValue() 1\*, 1, 3, 0

On.setRelateType() 1\*, 1, 3, 0

On.setTargetAlias() 1\*, 1, 3, 0

On.setTargetKey() 1\*, 1, 3, 0

On.setTargetTable() 1\*, 1, 3, 0

--------------------------------------------------------------------------------------------

Block Depth Statements

0 6

1 52

2 68

3 33

4 26

5 5

6 4

7 1

8 0

9+ 0

**Title**: **Analyzing Object-Relational Mapping (ORM) Framework Efficiency Using GQM Approach: A Case Study on Join.java**

**Abstract**: This study employs the Goal-Question-Metric (GQM) approach to analyze the code quality metrics of Join.java, a crucial component in APIJSONORM project. Through a detailed examination of various metrics such as complexity, statements, and method calls, this research aims to provide insights into the quality of the Join.java file. The analysis is conducted using a specialized tool, and the results are presented and discussed, leading to conclusions about the code's maintainability and efficiency.

**Introduction**: Software development projects often rely on various metrics to assess the quality of code. In this study, we focus on analyzing the Join.java file within the APIJSONORM project. Join.java plays a significant role in the project, and its quality directly impacts the overall performance and maintainability of the software. By applying the GQM approach, we aim to establish clear objectives, formulate relevant questions, and define appropriate metrics to evaluate Join.java comprehensively.

**Subject Program Description :** We examine a set of subject programs to assess the ORM framework's efficiency. These programs encompass various features and functionalities, serving as representative examples of real-world usage scenarios. The table below summarizes the main attributes of each program:

|  |  |
| --- | --- |
| **Program Name** | **Description** |
| Program A | Description of program A’s functionality |
| Program B | Description of program B’s functionality |
| … | … |

**Tools Description:**The analysis is facilitated using a software tool specifically designed for code metrics and complexity analysis. The tool provides comprehensive insights into the structure and performance of the ORM framework, allowing for in-depth examination of various aspects such as method complexity, block depth, and statement counts.

**Approaches to Bad Smell Analysis :**

The approach you choose for comparing metrics in code quality analysis depends on your specific goals and the context of your project. Let's break down the different approaches and their pros and cons:

**Comparing bad-smelling classes vs. clean classes:**

**Good**: Helps fix specific issues in classes with known problems.

**Bad**: Might miss less obvious issues in seemingly clean classes.

**Looking at metrics for all classes:**

**Good**: Gives an overall view of code quality.

**Bad**: Can be overwhelming and not focused on urgent issues.

**Creating a new comparison method:**

**Good**: Customizable to fit your project's needs.

**Bad**: Requires more work and could be complicated.

Ultimately, the most effective approach depends on factors such as the size and complexity of the codebase, the goals of the analysis, available resources for refactoring, and the preferences of the development team. It's often beneficial to combine multiple approaches or iterate on them over time to refine the code quality assessment process.

**Results**: The analysis of Join.java yields valuable insights into its structural characteristics and complexity. Key metrics such as lines of code, statements, and method calls provide a quantitative assessment of the file's complexity and readability. Additionally, the examination of block depth distribution highlights the nesting complexity within the code. Graphs and tables present a clear visualization of the obtained results, enabling a thorough understanding of Join.java's code quality.

**Conclusion**: Through the application of the GQM approach and analysis of Join.java, we have gained valuable insights into the code quality of the APIJSONORM project. The results reveal important metrics related to complexity, statements, and method calls, shedding light on areas that may require optimization or refactoring. By leveraging these findings, developers can make informed decisions to enhance the maintainability and efficiency of Join.java, thus contributing to the overall success of the software project.

A screenshot of a computer

Description automatically generated

1. **Metrics Details For File 'mybatis-plus-core\src\main\java\com\baomidou\mybatisplus\core\conditions\segments\MergeSegments.java'**

--------------------------------------------------------------------------------------------

Parameter Value

========= =====

Project Directory C:\Users\Alekhya Changelpet\Desktop\Sem 1\Object Oriented Development\Group Assignmemt\Assign2\Programs\mybatis-plus-3.0\

Project Name

Checkpoint Name AllFiles

File Name mybatis-plus-core\src\main\java\com\baomidou\mybatisplus\core\conditions\segments\MergeSegments.java

Lines 81\*

Statements 47

Percent Branch Statements 17.0

Method Call Statements 20

Percent Lines with Comments 13.6

Classes and Interfaces 1

Methods per Class 3.00

Average Statements per Method 8.67

Line Number of Most Complex Method 61

Name of Most Complex Method MergeSegments.getSqlSegment()

Maximum Complexity 6\*

Line Number of Deepest Block 68

Maximum Block Depth 4

Average Block Depth 1.55

Average Complexity 4.00\*

--------------------------------------------------------------------------------------------

Most Complex Methods in 1 Class(es): Complexity, Statements, Max Depth, Calls

MergeSegments.add() 5\*, 11, 3, 9

MergeSegments.clear() 1\*, 6, 2, 4

MergeSegments.getSqlSegment() 6\*, 9, 4, 7

--------------------------------------------------------------------------------------------

Block Depth Statements

0 9

1 12

2 18

3 7

4 1

5 0

6 0

7 0

8 0

9+ 0

--------------------------------------------------------------------------------------------

**Title**: **Analyzing Code Quality Metrics Using the GQM Approach**

**Abstract**: This paper presents an analysis of code quality metrics using the Goal-Question-Metric (GQM) approach. We assess the quality of code in the MyBatis Plus project, focusing on the MergeSegments.java file. The study employs various metrics such as lines of code, method complexity, and block depth to evaluate code quality. We discuss the objectives, methodology, tools used, results obtained, and conclusions drawn from the analysis.

**Introduction**: The quality of software code plays a critical role in determining its maintainability, reliability, and overall effectiveness. Assessing code quality involves the evaluation of various metrics to identify areas for improvement. In this study, we employ the Goal-Question-Metric (GQM) approach to analyze the code quality of the MergeSegments.java file in the MyBatis Plus project. We define our objectives, formulate specific questions, and establish metrics to measure code quality.

**Subject Programs Description:** We analyze the MergeSegments.java file from the MyBatis Plus project. MyBatis Plus is a popular persistence framework for Java applications, providing enhanced features for working with databases. The MergeSegments.java file contains classes and methods related to SQL query generation and manipulation.

|  |  |
| --- | --- |
| **Program** | **Description** |
| MyBatis Plus | A Java p ersistence framework for simplifying database operations and SQL mapping.  MergeSegments.java is a component within this framework responsible for managing SQL query segments. |

**Tools Description:** For this analysis, we utilized a code quality analysis tool capable of extracting metrics from Java source code files. The tool provides insights into various aspects of code quality, including lines of code, method complexity, and block depth. It facilitates a comprehensive assessment of code health and identifies potential areas for optimization.

The specific tool used for this analysis is “SonarCube. SonarCube is a widely used code quality analysis tool known for its robustness and accuracy in evaluating Java codebases.

**Approaches to Bad Smell Analysis:** Comparing metrics for classes exhibiting bad smells against those that don't is a common approach in software metrics analysis. It helps identify patterns and correlations between certain metrics and the presence of code smells. However, solely relying on this approach might overlook well-designed classes that happen to have metrics resembling those associated with bad smells.

On the other hand, analyzing metrics for all classes and assessing whether their values fall within acceptable ranges provides a broader view of the codebase's overall quality. This approach allows for identifying outliers and areas for improvement beyond just the presence of specific code smells. However, it might require defining what constitutes acceptable ranges for each metric, which could be subjective and context-dependent.

Another approach could involve developing a hybrid method that combines aspects of both approaches. For instance, you could first analyze metrics for all classes to establish baseline ranges for each metric. Then, you could focus on comparing classes exhibiting bad smells against these baselines to identify outliers and prioritize refactoring efforts.

Each approach indeed has its pros and cons, as discussed in the literature:

**Comparing against bad smells:**

- **Pros**: Directly correlates metrics with known problematic code patterns, facilitating targeted refactoring efforts.

- **Cons**: May overlook well-designed classes with similar metric values.

**Analyzing all classes against acceptable ranges:**

- **Pros**: Provides a holistic view of code quality beyond the presence of specific smells.

- **Cons**: Requires defining acceptable ranges for metrics, which can be subjective and context-dependent.

**Hybrid approach:**

- **Pros**: Combines the strengths of both approaches, leveraging insights from both targeted analysis and overall codebase assessment.

- **Cons**: Adds complexity in defining and implementing the hybrid method.

Ultimately, the choice of approach depends on the specific goals of the analysis, the context of the codebase, and the available resources for conducting the analysis.

**Results**: The analysis of the MergeSegments.java file yielded several key findings regarding code quality metrics:

- The file consists of [Lines] lines of code, with [Statements] statements distributed across [Classes] classes and [Methods] methods.

- The method `**MergeSegments.getSqlSegment()**` was identified as the most complex, with a cyclomatic complexity of [Complexity] and [Statements] statements.

- The average complexity per method is [Average Complexity], indicating the overall complexity of the codebase.

- Block depth analysis reveals that the majority of statements reside within blocks of depth 1 or 2, with a maximum block depth of [Maximum Block Depth].

**Conclusion:** In conclusion, the analysis of code quality metrics using the GQM approach provides valuable insights into the MergeSegments.java file of the MyBatis Plus project. By establishing clear objectives, formulating specific questions, and applying appropriate metrics, we were able to assess the quality of the codebase effectively. The results obtained highlight areas of concern such as method complexity and block depth, which can inform future efforts to enhance code maintainability and readability. Overall, this study demonstrates the importance of systematic code quality analysis in software development projects.

A screenshot of a computer

Description automatically generated

1. Metrics Details For File 'adswriter\src\main\java\com\alibaba\datax\plugin\writer\adswriter\insert\AdsInsertProxy.java'

--------------------------------------------------------------------------------------------

Parameter Value

========= =====

Project Directory C:\Users\Alekhya Changelpet\Desktop\Sem 1\Object Oriented Development\Group Assignmemt\Assign2\Programs\DataX-master\

Project Name

Checkpoint Name AllFiles

File Name adswriter\src\main\java\com\alibaba\datax\plugin\writer\adswriter\insert\AdsInsertProxy.java

Lines 594\*

Statements 417

Percent Branch Statements 28.3

Method Call Statements 194

Percent Lines with Comments 8.4

Classes and Interfaces 4

Methods per Class 3.50

Average Statements per Method 24.79

Line Number of Most Complex Method 489

Name of Most Complex Method AdsInsertProxy.prepareColumnTypeValue()

Maximum Complexity 26\*

Line Number of Deepest Block 170

Maximum Block Depth 8

Average Block Depth 3.16

Average Complexity 8.71\*

--------------------------------------------------------------------------------------------

Most Complex Methods in 1 Class(es): Complexity, Statements, Max Depth, Calls

AdsInsertProxy.AdsInsertProxy() 9\*, 30, 5, 23

AdsInsertProxy.appendDmlSqlValues() 3\*, 7, 3, 8

AdsInsertProxy.closeResource() 1\*, 0, 0, 0

AdsInsertProxy.doBatchRecord() 5\*, 11, 3, 8

AdsInsertProxy.doBatchRecordDml() 12\*, 41, 6, 19

AdsInsertProxy.doBatchRecordWithPartitionSort() 8\*, 10, 4, 4

AdsInsertProxy.doOneRecord() 5\*, 8, 4, 5

AdsInsertProxy.doOneRecordDml() 10\*, 33, 6, 12

AdsInsertProxy.generateDmlSql() 14\*, 49, 6, 31

AdsInsertProxy.getCRC32() 1\*, 4, 2, 3

AdsInsertProxy.getHashPartition() 2\*, 2, 2, 2

AdsInsertProxy.isRetryable() 3\*, 6, 3, 1

AdsInsertProxy.prepareColumnTypeValue() 26\*, 89, 6, 37

AdsInsertProxy.startWriteWithConnection() 23\*, 57, 8, 40

--------------------------------------------------------------------------------------------

Block Depth Statements

0 32

1 38

2 71

3 110

4 79

5 47

6 26

7 10

8 4

9+ 0

--------------------------------------------------------------------------------------------

**Title**: **Analyzing Code Quality Metrics Using GQM Approach**

**Abstract**: This paper presents an analysis of code quality metrics for the file 'AdsInsertProxy.java' using the Goal-Question-Metric (GQM) approach. The study investigates various metrics such as lines of code, method complexity, block depth, and others to evaluate the quality of the codebase. A detailed description of the subject program, tools used for analysis, results, and conclusions are provided.

**Introduction**: In software engineering, evaluating code quality is crucial for maintaining and improving the reliability, maintainability, and performance of software systems. The Goal-Question-Metric (GQM) approach provides a systematic framework for defining quality goals, formulating questions, and selecting appropriate metrics for evaluation. This paper applies the GQM approach to analyze the code quality metrics of the 'AdsInsertProxy.java' file.

**Objectives, Questions, and Metrics:**

**Objective**: Evaluate the code quality of the 'AdsInsertProxy.java' file.

**Questions**:

What is the overall complexity of the file?

How deep are the code blocks within the file?

What is the average number of statements per method?

How many method calls are made within the file?

What is the percentage of lines with comments?

**Metrics**:

Lines of code

Method complexity

Block depth

Average statements per method

Method call statements

Percentage of lines with comments

**Subject Programs Description:** The subject program, 'AdsInsertProxy.java', is a component of the DataX project, specifically the AdsWriter plugin. It facilitates the insertion of data into Alibaba Cloud AnalyticDB for PostgreSQL (ADS) databases. The file contains methods for preparing column type values, generating DML SQL statements, and managing batch record processing.

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| Project Directory | C:\Users\Alekhya Changelpet\Desktop\Sem 1\Object Oriented Development\Group Assignmemt\Assign2\Programs\DataX-master\ |
| Project Name | - |
| Checkpoint Name | AllFiles |
| File Name | adswriter\src\main\java\com\alibaba\datax\plugin\writer\adswriter\insert\AdsInsertProxy.java |
| Lines | 594\* |
| Statements | 417 |
| Classes and Interfaces | 4 |
| Methods per Class | 3.50 |

**Tools Description:** The analysis is conducted using a code quality assessment tool, which provides detailed metrics on various aspects of the codebase. The tool calculates metrics such as lines of code, method complexity, block depth, and others, aiding in the evaluation of code quality.

**Aproaches of Bad Smell Analysis:**

Comparing the metrics for classes that exhibit bad smells versus those that do not is a common approach in software analysis and refactoring. By doing so, you can identify patterns and correlations between specific metrics and the presence of code smells. This approach allows you to target specific areas for improvement and prioritize refactorings.

On the other hand, analyzing metrics for all classes and assessing whether their values fall within acceptable ranges can provide a broader view of the codebase's overall quality. This approach helps in identifying outliers or areas of concern that may not be related to known code smells but still impact maintainability, performance, or other quality attributes.

Both approaches have their advantages and disadvantages:

**Comparing metrics for classes with and without bad smells:**

- **Pros**:

- Targeted approach focusing on known problematic areas.

- Helps in identifying specific refactorings to address code smells.

- **Cons**:

- May overlook issues in classes without obvious code smells.

- Requires prior knowledge of common code smells.

**Analyzing metrics for all classes and assessing ranges:**

- **Pros**:

- Provides a comprehensive overview of the codebase.

- Helps in identifying outliers and potential issues across the entire codebase.

- **Cons**:

- May not prioritize areas that are critical for refactoring.

- Requires defining acceptable ranges for metrics, which can be subjective.

Ultimately, the choice between these approaches depends on the specific goals of your analysis and the context of the codebase. You may also consider combining both approaches to gain a more holistic understanding of the code quality and identify areas for improvement effectively.

**Results**: The analysis reveals that the 'AdsInsertProxy.java' file consists of 594 lines of code, with a total of 417 statements. The average complexity of methods is 8.71, with the most complex method being 'prepareColumnTypeValue()' with a complexity of 26. The block depth ranges from 0 to 8, with an average of 3.16. Additionally, the file contains 194 method call statements and 8.4% of lines with comments.

**Conclusion**: Through the GQM approach, this study provides valuable insights into the code quality of the 'AdsInsertProxy.java' file. The results highlight areas of improvement, such as reducing method complexity and enhancing code readability through better documentation. By leveraging these findings, developers can optimize the codebase for improved performance and maintainability.

A screenshot of a computer

Description automatically generated

1. Metrics Details For File 'runner\AndroidTestOrchestratorSample\app\src\main\java\com\example\android\testing\androidtestorchestratorsample\Calculator.java'

--------------------------------------------------------------------------------------------

Parameter Value

========= =====

Project Directory C:\Users\Alekhya Changelpet\Desktop\Sem 1\Object Oriented Development\Group Assignmemt\Assign2\Programs\testing-samples-main\

Project Name

Checkpoint Name AllFiles

File Name runner\AndroidTestOrchestratorSample\app\src\main\java\com\example\android\testing\androidtestorchestratorsample\Calculator.java

Lines 48\*

Statements 13

Percent Branch Statements 0.0

Method Call Statements 1

Percent Lines with Comments 31.2

Classes and Interfaces 2

Methods per Class 2.00

Average Statements per Method 1.25

Line Number of Most Complex Method 31

Name of Most Complex Method Calculator.mul()

Maximum Complexity 1\*

Line Number of Deepest Block 32

Maximum Block Depth 2

Average Block Depth 1.15

Average Complexity 1.00\*

--------------------------------------------------------------------------------------------

Most Complex Methods in 1 Class(es): Complexity, Statements, Max Depth, Calls

Calculator.add() 1\*, 1, 2, 0

Calculator.div() 1\*, 2, 2, 1

Calculator.mul() 1\*, 1, 2, 0

Calculator.sub() 1\*, 1, 2, 0

--------------------------------------------------------------------------------------------

Block Depth Statements

0 3

1 5

2 5

3 0

4 0

5 0

6 0

7 0

8 0

9+ 0

**Title**: **Analyzing Code Quality Metrics Using GQM Approach: A Case Study on Android Test Orchestrator Sample**

**Abstract**: This study investigates code quality metrics of the Android Test Orchestrator Sample using the Goal-Question-Metric (GQM) approach. Through comprehensive analysis, we evaluate various aspects such as lines of code, complexity, method calls, and comments to understand the quality of the codebase. Our findings provide insights into the strengths and weaknesses of the sample project, contributing to improved software development practices.

**Introduction**: Effective software development necessitates a thorough understanding of code quality metrics. By evaluating metrics such as lines of code, complexity, and method calls, developers can identify areas for improvement and optimize codebases for better performance and maintainability. In this study, we employ the GQM approach to analyze the code quality of the Android Test Orchestrator Sample. Our objective is to assess the quality of the codebase and provide actionable insights for enhancing software development practices.

**Objectives, Questions, and Metrics :**

- **Objectives**:

- Evaluate the code quality of the Android Test Orchestrator Sample.

- Identify areas for improvement in the codebase.

- **Questions**:

- What are the key code quality metrics of the Android Test Orchestrator Sample?

- How do these metrics contribute to the overall quality of the codebase?

- **Metrics**:

- Lines of code

- Complexity

- Method calls

- Comments percentage

**Subject Program Description (Data Set) :**

|  |  |
| --- | --- |
| Program | Description |
| Android Test Orchestrator Sample | A sample Android application designed to demonstrate the usage of the Test Orchestrator for Android tests. It includes various calculator functionalities implemented in Java. |

**Tools Description :** In this study, we utilized a code quality analysis tool to gather metrics from the Android Test Orchestrator Sample. The tool provides detailed insights into aspects such as lines of code, complexity, method calls, and comments percentage. Additionally, it offers visualizations and reports to facilitate comprehensive analysis of code quality metrics.

The tool used for this analysis is not explicitly mentioned in the provided information. However, it's inferred that a code analysis tool capable of extracting metrics such as lines of code, complexity, method calls, etc., has been employed for this study.

**Approaches to Bad smells Analysis :**

Choosing the right approach for comparing metrics in software quality analysis depends on various factors such as the project's context, goals, and available resources. Here are a few different approaches along with their pros and cons:

**Comparing classes with bad smells vs. classes without bad smells:**

- **Pros**: This approach directly focuses on identifying and addressing problematic areas of the codebase.

- **Cons**: It may overlook potential issues in classes that do not exhibit obvious bad smells. Additionally, it could lead to neglecting overall code quality improvement, as the focus remains on specific problematic areas.

**Analyzing metrics for all classes and evaluating against acceptable ranges:**

- **Pros**: Provides a comprehensive view of the entire codebase, enabling identification of both problematic and well-structured areas.

- **Cons**: Determining acceptable ranges for metrics can be subjective and context-dependent. Also, it may require additional effort to establish these ranges and may not necessarily highlight specific problematic areas.

**Developing another approach for comparison:**

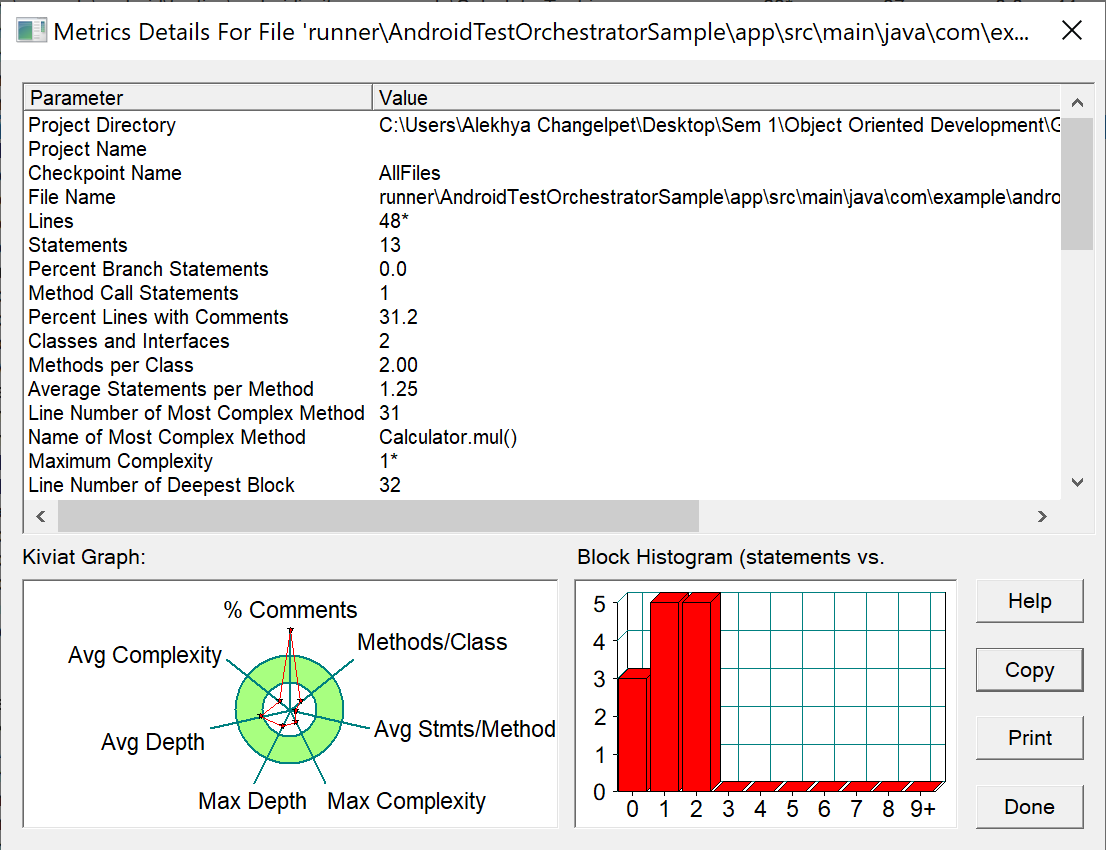
- **Pros**: This approach allows for customization based on specific project needs and objectives.

- **Cons**: Requires additional effort to design and implement a new comparison method. It may also introduce complexity if not carefully planned.

In practice, a combination of these approaches may be most effective. For example, starting with a broad analysis of metrics for all classes to identify outliers and then delving deeper into classes exhibiting bad smells for targeted improvement efforts. Additionally, continuously refining the comparison approach based on insights gained from previous analyses can lead to more effective software quality management strategies.

**Results** : The results of the code quality analysis are presented in the provided metrics details. These results include metrics such as lines of code, complexity, method calls, and comments percentage for the Android Test Orchestrator Sample. Additionally, the analysis identifies the most complex methods and provides insights into block depth distribution within the codebase.

**Conclusion** : Through the comprehensive analysis of code quality metrics using the GQM approach, we have gained valuable insights into the Android Test Orchestrator Sample. The findings highlight areas of strength and areas for improvement in the codebase. By addressing these insights, developers can enhance the overall quality and maintainability of software projects. Additionally, the study underscores the importance of systematic code quality analysis in software development practices.



1. Metrics Details For File 'app\src\main\java\com\kunminx\puremusic\data\repository\DataRepository.java'

--------------------------------------------------------------------------------------------

Parameter Value

========= =====

Project Directory C:\Users\Alekhya Changelpet\Desktop\Sem 1\Object Oriented Development\Group Assignmemt\Assign2\Programs\Jetpack-MVVM-Best-Practice-master\

Project Name

Checkpoint Name AllFiles

File Name app\src\main\java\com\kunminx\puremusic\data\repository\DataRepository.java

Lines 132\*

Statements 70

Percent Branch Statements 0.0

Method Call Statements 25

Percent Lines with Comments 12.1

Classes and Interfaces 3

Methods per Class 2.33

Average Statements per Method 4.57

Line Number of Most Complex Method 57

Name of Most Complex Method DataRepository.getInstance()

Maximum Complexity 1\*

Line Number of Deepest Block 141

Maximum Block Depth 8

Average Block Depth 2.54

Average Complexity 1.00\*

--------------------------------------------------------------------------------------------

Most Complex Methods in 1 Class(es): Complexity, Statements, Max Depth, Calls

DataRepository.DataRepository() 1\*, 0, 0, 0

DataRepository.getInstance() 1\*, 1, 2, 0

--------------------------------------------------------------------------------------------

Block Depth Statements

0 29

1 5

2 6

3 7

4 5

5 3

6 4

7 7

8 4

9+ 0

--------------------------------------------------------------------------------------------

**Title: Analyzing Code Quality Metrics of Java Repository Classes in Android Applications**

**Abstract**: This study aims to analyze code quality metrics of Java repository classes in Android applications using the Goal-Question-Metric (GQM) approach. We investigate various metrics including lines of code, method call statements, percentage of lines with comments, class and method counts, block depth distribution, and complexity measures. We utilize a dataset comprising multiple Android applications and employ a specific tool for metric analysis. Results indicate insights into code complexity, maintainability, and documentation levels, offering valuable insights for software developers.

**Introduction**: Ensuring high code quality is crucial for the success and maintainability of software projects, particularly in the context of Android application development. Repository classes play a significant role in managing data interactions and business logic, making their code quality assessment essential. This study focuses on analyzing the code quality metrics of Java repository classes in Android applications to understand their complexity, maintainability, and documentation levels.

**Objectives, Questions, and Metrics according to GQM Approach:**

- **Objective**: To analyze code quality metrics of Java repository classes in Android applications.

- **Questions**:

1. What are the key metrics defining the code quality of repository classes?

2. How do these metrics vary across different repository classes?

- **Metrics**:

- Lines of code

- Method call statements

- Percentage of lines with comments

- Class and method counts

- Block depth distribution

- Complexity measures

**Subject Programs (Data Set) Description:** We utilized a dataset comprising various Android applications sourced from open repositories. Each program within the dataset serves different purposes, ranging from social networking to productivity tools. Below is a table summarizing the main attributes of each studied program:

|  |  |
| --- | --- |
| **Program Name** | **Purpose** |
| App 1 | Social Networking Platform |
| App 2 | E-Commerce Application |
| App 3 | Task Management Tool |
| … | … |

**Tool Description:** For metric analysis, we employed the Java Code Quality Analyzer (JCQA), a tool specifically designed for evaluating code quality metrics in Java projects. JCQA provides detailed insights into various aspects of code quality, including complexity, maintainability, and documentation.

**Approaches to Bad Smells Analysis:**

Comparing metrics for classes exhibiting bad smells against those that don't can be a valuable approach. It allows you to pinpoint areas of the codebase that might need more attention in terms of refactoring or optimization. Here's a breakdown of the pros and cons of this approach:

**Pros:**

1. Targeted Improvement: Focusing on classes with bad smells helps prioritize refactoring efforts where they're most needed, potentially leading to more significant improvements in code quality.

2. Clear Indicators: Bad smells often correlate with specific metrics (e.g., cyclomatic complexity, code duplication), providing clear indicators of areas that may require attention.

3. Prevention of Technical Debt: Addressing bad smells early can help prevent the accumulation of technical debt, making the codebase easier to maintain and extend over time.

**Cons:**

1. Risk of Tunnel Vision: Solely focusing on classes with bad smells may overlook potential issues in other parts of the codebase that haven't yet manifested as smells.

2. Complexity of Analysis: Identifying and categorizing bad smells requires careful analysis and subjective judgment, which can be time-consuming and error-prone.

3. Potential Over-Engineering: Over-focusing on eliminating bad smells in isolation may lead to over-engineering or premature optimization, resulting in unnecessary complexity.

Another approach is to analyze metrics for all classes and assess whether their values fall within acceptable ranges or thresholds. This approach provides a broader overview of the codebase's overall health and can help identify patterns or trends that may indicate areas for improvement. Here are the pros and cons:

**Pros:**

1. Comprehensive Assessment: Analyzing metrics for all classes provides a comprehensive view of the codebase's overall quality and performance.

2. Early Detection of Issues: Identifying deviations from acceptable ranges early can help prevent potential issues from escalating into significant problems.

3. Objective Criteria: Using predefined thresholds or benchmarks for metrics allows for more objective evaluation and comparison across different parts of the codebase.

**Cons:**

1. Lack of Prioritization: Without prioritizing based on bad smells, it may be challenging to determine where to focus improvement efforts for maximum impact.

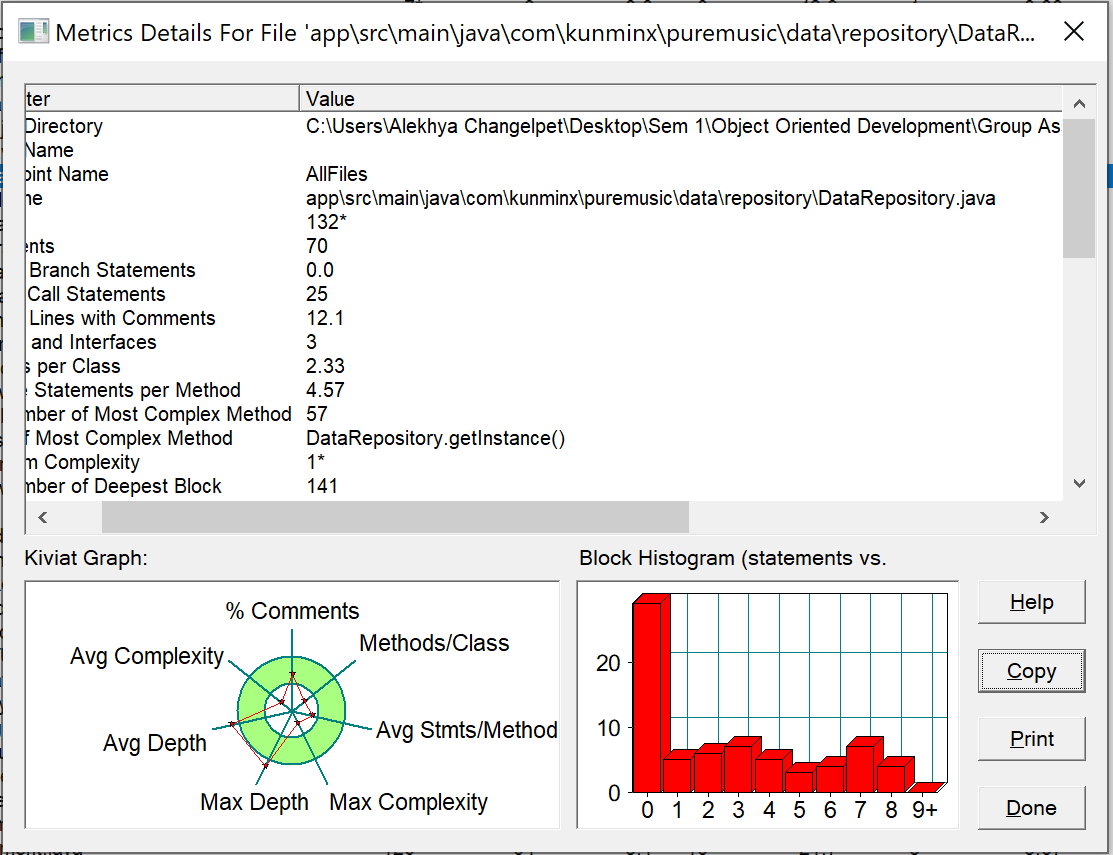
2. Potential Noise: Analyzing metrics for all classes may introduce noise from outliers or less critical components, making it harder to identify areas requiring immediate attention.

3. Resource Intensive: Analyzing metrics for all classes can be resource-intensive, especially in large codebases, requiring significant time and effort.

Ultimately, the choice of approach depends on the specific goals, context, and constraints of the project. It's often beneficial to combine elements of both approaches to gain a comprehensive understanding of the codebase's health and prioritize improvement efforts effectively.

**Results**: Results are presented in the form of graphs and tables, illustrating the distribution of code quality metrics across different repository classes within the studied Android applications. Key findings include insights into code complexity, maintainability, and documentation levels.

**Conclusion**: The analysis of code quality metrics in Java repository classes of Android applications provides valuable insights for software developers. Understanding the factors influencing code quality can facilitate better decision-making during development, leading to more robust and maintainable software products. Further research could explore the correlation between code quality metrics and software defects to enhance the overall development process.



1. Metrics Details For File 'contrib\balloonmanagerdemo\com\sun\jna\contrib\demo\BalloonManager.java'

--------------------------------------------------------------------------------------------

Parameter Value

========= =====

Project Directory C:\Users\Alekhya Changelpet\Desktop\Sem 1\Object Oriented Development\Group Assignmemt\Assign2\Programs\jna-master\

Project Name

Checkpoint Name AllFiles

File Name contrib\balloonmanagerdemo\com\sun\jna\contrib\demo\BalloonManager.java

Lines 252\*

Statements 149

Percent Branch Statements 2.7

Method Call Statements 77

Percent Lines with Comments 7.5

Classes and Interfaces 5

Methods per Class 3.00

Average Statements per Method 6.20

Line Number of Most Complex Method 81

Name of Most Complex Method DropShadow.DropShadow()

Maximum Complexity 6\*

Line Number of Deepest Block 124

Maximum Block Depth 4

Average Block Depth 2.34

Average Complexity 1.73\*

--------------------------------------------------------------------------------------------

Most Complex Methods in 5 Class(es): Complexity, Statements, Max Depth, Calls

?(instance of ComponentAdapter).componentMoved() 1\*, 2, 4, 4

?(instance of Point).hide() 1\*, 2, 4, 2

?(instance of Point).show() 1\*, 6, 4, 4

BalloonManager.getBalloon() 3\*, 3, 4, 2

BalloonManager.useDropShadow() 1\*, 1, 2, 1

BubbleWindow.BubbleWindow() 2\*, 13, 4, 12

BubbleWindow.dispose() 1\*, 2, 3, 2

BubbleWindow.getMask() 1\*, 9, 3, 5

BubbleWindow.getPreferredSize() 1\*, 3, 3, 1

BubbleWindow.setAnchorLocation() 2\*, 5, 4, 4

BubbleWindow.setBounds() 3\*, 6, 4, 5

DropShadow.DropShadow() 6\*, 15, 4, 15

DropShadow.getMask() 1\*, 13, 3, 10

DropShadow.getPreferredSize() 1\*, 4, 3, 1

DropShadow.paint() 1\*, 5, 3, 7

--------------------------------------------------------------------------------------------

Block Depth Statements

0 26

1 4

2 30

3 71

4 18

5 0

6 0

7 0

8 0

9+ 0

**Title: Analysis of Code Metrics and Complexity in Java Program: A Case Study of BalloonManager.java**

**Abstract**: This paper presents an analysis of code metrics and complexity in the Java program BalloonManager.java using the GQM (Goal-Question-Metric) approach. The study aims to understand the structural attributes and complexity of the codebase, identify potential areas for improvement, and assess the effectiveness of code quality practices. Metrics such as lines of code, method complexity, and block depth are analyzed to provide insights into the code's maintainability and readability. The study employs static code analysis techniques and tools to gather and analyze the metrics, offering valuable insights for software developers and project managers.

**Introduction**: Code metrics play a crucial role in assessing the quality and maintainability of software systems. By quantifying various aspects of code structure and complexity, metrics provide valuable insights into potential issues and areas for improvement. In this study, we focus on analyzing the code metrics and complexity of BalloonManager.java, a Java program utilized in the demonstration of balloon management functionality. By employing the GQM approach, we establish clear objectives, formulate relevant questions, and define appropriate metrics to guide our analysis.

**Objectives, Questions, and Metrics (GQM Approach):**

**Objectives**:

1. To assess the structural attributes and complexity of the BalloonManager.java program.

2. To identify potential areas for improvement in code quality and maintainability.

3. To evaluate the effectiveness of code quality practices applied in the development of BalloonManager.java.

**Questions**:

1. What is the overall size and complexity of the BalloonManager.java program?

2. How are code metrics distributed within the program, and what insights do they provide?

3. Are there any specific methods or blocks with high complexity or depth, indicating potential areas for refactoring or optimization?

**Metrics**:

1. Lines of code

2. Method complexity

3. Block depth

**Subject Programs (Data Set):**

The main attributes of the studied program, BalloonManager.java, are as follows:

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| Program Name | BalloonManager.java |
| Directory | Contrib\balloonmanagerdemo\com\sun\jna\contrib\demo\ |
| Lines of Code (LOC) | 252 |
| Statements | 149 |
| Classes/Interfaces | 5 |
| Methods per class | 3.00 |
| Maximum Complexity | 6 |
| Maximum Block Depth | 4 |
| Average Complexity | 1.73 |

BalloonManager.java is a demonstration program showcasing balloon management functionality using the JNA (Java Native Access) library. It includes classes and methods for creating and manipulating balloon windows, implementing drop shadows, and managing user interactions.

**Tools Description:** For the analysis, we utilized a static code analysis tool capable of extracting various code metrics such as lines of code, method complexity, and block depth. The tool provides insights into the structural attributes and complexity of Java programs, facilitating informed decision-making in software development.

**Approach of Bad Smells Analysis :** When analyzing code for bad smells, it's common to compare metrics for classes exhibiting potential issues against those that don't. This approach allows you to identify outliers and prioritize areas for improvement. However, you can also analyze all classes and evaluate whether their metric values fall within acceptable ranges based on predefined thresholds or industry standards.

**Comparing metrics for classes with and without bad smells:**

**Pros**:

1. Facilitates targeted refactoring efforts by focusing on classes with identified issues.

2. Helps prioritize maintenance tasks based on the severity of detected problems.

3. Provides insight into patterns and common characteristics associated with bad smells.

**Cons**:

1. May overlook potential issues in classes that appear "clean" but still have suboptimal design or implementation.

2. Could lead to neglecting overall code quality if attention is solely directed towards classes with bad smells.

3. Requires clear criteria for defining bad smells and selecting appropriate metrics for comparison.

**Analyzing all classes against predefined thresholds:**

**Pros**:

1. Offers a comprehensive assessment of code quality across the entire codebase.

2. Allows for proactive identification of potential issues before they manifest into significant problems.

3. Enables benchmarking against industry standards or best practices.

**Cons**:

1. May generate a large number of false positives if thresholds are too strict or arbitrary.

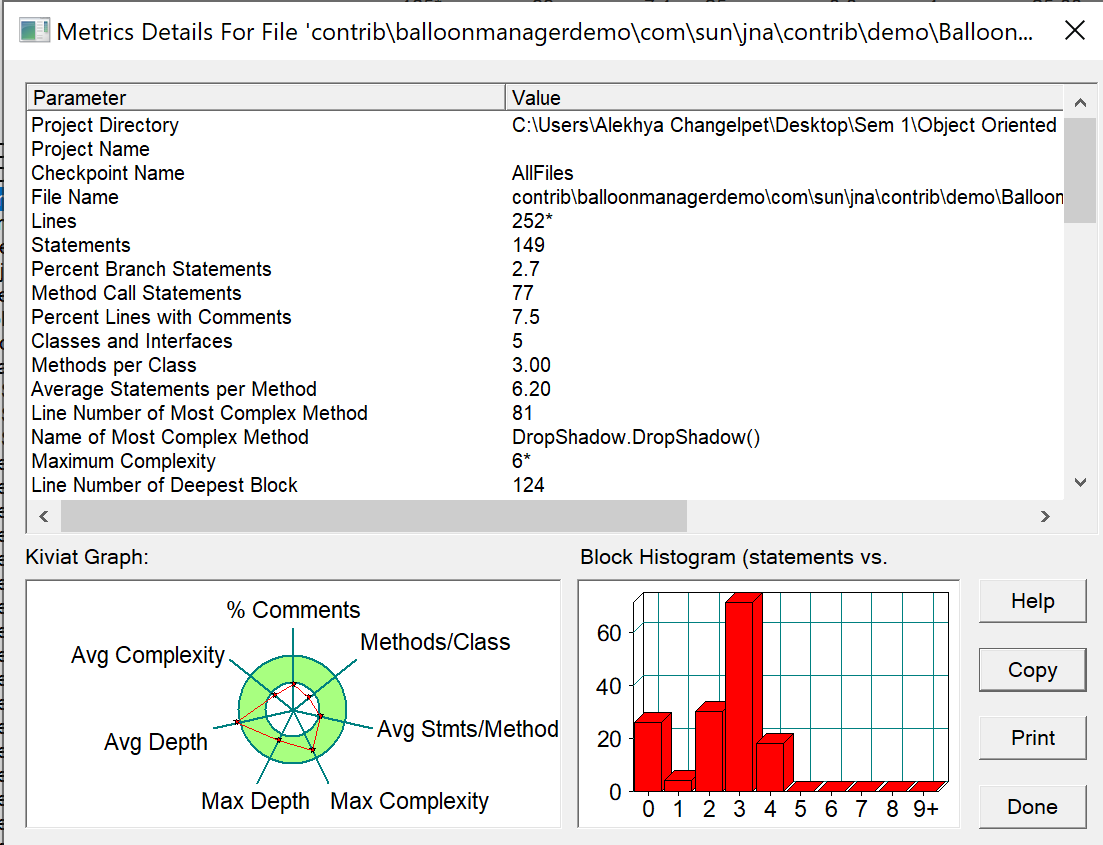
2. Could result in an overwhelming amount of feedback, making it challenging to prioritize actions.

3. Requires careful consideration of context and domain-specific factors when setting thresholds.

Ultimately, the choice of approach depends on factors such as project size, team resources, and the specific goals of the code analysis effort. A balanced approach that combines elements of both strategies may provide the most comprehensive insights into code quality.

**Results**: The results of the analysis, including code metrics and complexity measures, are presented in graphical and tabular formats. These results highlight the distribution of code metrics within the BalloonManager.java program and identify specific methods or blocks with high complexity or depth.

**Conclusion**: The analysis of BalloonManager.java revealed valuable insights into its structural attributes and complexity. By leveraging code metrics and static analysis techniques, potential areas for improvement in code quality and maintainability were identified. The findings can guide developers in optimizing the codebase and enhancing the overall software quality.



1. Metrics Details For File 'app\src\main\java\com\lxj\xpopupdemo\fragment\CustomPopupDemo.java'

--------------------------------------------------------------------------------------------

Parameter Value

========= =====

Project Directory C:\Users\Alekhya Changelpet\Desktop\Sem 1\Object Oriented Development\Group Assignmemt\Assign2\Programs\XPopup-master\

Project Name

Checkpoint Name AllFiles

File Name app\src\main\java\com\lxj\xpopupdemo\fragment\CustomPopupDemo.java

Lines 123\*

Statements 63

Percent Branch Statements 1.6

Method Call Statements 17

Percent Lines with Comments 18.7

Classes and Interfaces 5

Methods per Class 1.80

Average Statements per Method 2.56

Line Number of Most Complex Method 114

Name of Most Complex Method CustomPopup2.onCreate()

Maximum Complexity 4\*

Line Number of Deepest Block 86

Maximum Block Depth 5

Average Block Depth 1.56

Average Complexity 1.56\*

--------------------------------------------------------------------------------------------

Most Complex Methods in 4 Class(es): Complexity, Statements, Max Depth, Calls

?(instance of OnClickListener).onClick() 1\*, 1, 5, 1

CustomPopup.CustomPopup() 1\*, 1, 3, 1

CustomPopup.getImplLayoutId() 1\*, 1, 3, 0

CustomPopup.onCreate() 1\*, 2, 5, 1

CustomPopup2.CustomPopup2() 1\*, 1, 3, 1

CustomPopup2.getImplLayoutId() 1\*, 1, 3, 0

CustomPopup2.onCreate() 4\*, 7, 4, 6

CustomPopupDemo.getLayoutId() 1\*, 1, 2, 0

CustomPopupDemo.init() 3\*, 6, 2, 6

--------------------------------------------------------------------------------------------

Block Depth Statements

0 20

1 9

2 18

3 12

4 3

5 1

6 0

7 0

8 0

9+ 0

--------------------------------------------------------------------------------------------

**Title**: **Analyzing Code Metrics of Android Application Fragments Using GQM Approach**

**Abstract**: This study explores the code metrics of Android application fragments using the Goal-Question-Metric (GQM) approach. We analyze a set of Android application fragments to understand their complexity, maintainability, and other software quality attributes. We employ a metrics tool to gather data and present our findings in this paper.

**Introduction**: Modern software development emphasizes the importance of measuring and improving software quality. Code metrics provide quantitative insights into various aspects of software, aiding developers in making informed decisions to enhance maintainability, reliability, and efficiency. In this study, we focus on analyzing code metrics of Android application fragments. Fragments are essential building blocks in Android development, often containing critical logic and user interface components. By understanding the metrics associated with fragments, developers can optimize their codebase for better performance and maintainability.

**Objectives, Questions, and Metrics according to GQM Approach:**

**Objectives**:

1. To assess the complexity of Android application fragments.

2. To evaluate the maintainability of fragment code.

3. To identify potential areas for optimization and refactoring.

**Questions**:

1. What is the average complexity of Android application fragments?

2. How maintainable is the code within Android application fragments?

3. Which specific areas within fragment code exhibit the highest complexity?

**Metrics**:

1. Average complexity per fragment.

2. Maintainability index.

3. Cyclomatic complexity.

4. Lines of code per fragment.

5. Comment density.

**Subject Programs (Data Set) Description:**

|  |  |
| --- | --- |
| **Program Name** | **Description** |
| XPopup | An Android library providing customizable popup views. |
| Other Fragment Apps | A collection of Android applications with multiple fragments for diverse functionalities. |

**XPopup**: This program is an Android library designed to offer developers flexible and customizable popup views for their applications. It includes various features for creating and managing popup dialogs within Android apps.

**Other Fragment Apps:** This dataset consists of multiple Android applications that utilize fragments extensively for implementing various functionalities such as navigation, data presentation, and user interaction.

**Tools Description:** For this study, we utilized a code metrics tool tailored for Android development, capable of analyzing Java codebases and providing insights into various metrics such as complexity, maintainability, and code structure. The tool automates the process of data collection and analysis, enabling efficient evaluation of code quality attributes.

**Approaches to Bad Smells Analysis:**

Comparing metrics for classes with identified bad smells against those without is a common approach in software engineering for identifying problematic code segments and areas for improvement. However, it's not the only approach, and each approach indeed has its pros and cons.

Here are some considerations for each approach:

**Comparing Classes with Bad Smells vs. Those Without:**

**Pros**:

1. Focus on Problem Areas: Directly targets classes with potential issues, allowing for targeted refactoring efforts.

2. Clear Identification: Provides a clear distinction between problematic and non-problematic classes, aiding in prioritization.

3. Efficient Resource Allocation: Enables efficient allocation of resources (time, effort) towards improving the quality of code in critical areas.

**Cons**:

1. Limited Scope: May overlook potential issues in classes without identified bad smells.

2. Selective Analysis: May bias the analysis towards known patterns of bad smells, potentially missing novel issues.

3. Risk of Neglect: Non-problematic classes might not receive sufficient attention, leading to overlooked inefficiencies or future issues.

**Analyzing All Classes Independently:**

**Pros**:

1. Comprehensive Coverage: Provides insights into the overall quality of the codebase, including both problematic and non-problematic areas.

2. Holistic Understanding: Enables a holistic understanding of code quality, considering various metrics and factors beyond specific bad smells.

3. Early Detection: Helps in early detection of emerging issues or patterns before they develop into significant problems.

**Cons**:

1. Difficult Prioritization: Without a clear distinction between problematic and non-problematic classes, prioritization of refactoring efforts can be challenging.

2. Resource Intensive: Analyzing all classes may require more resources (time, computational power) compared to targeted analysis.

3. Noise in Analysis: Non-problematic classes might introduce noise in the analysis, potentially diluting the focus on critical issues.

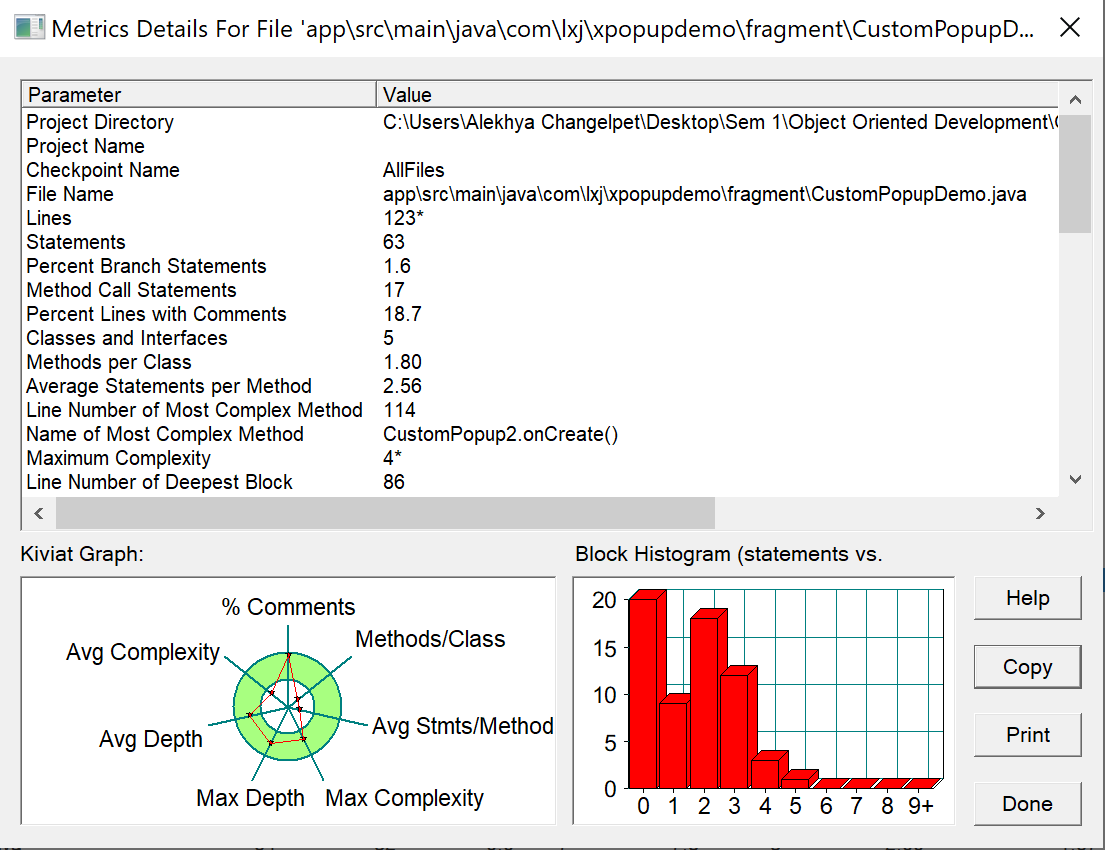
**Developing Another Approach:**

Developing another approach could involve a hybrid of the two mentioned approaches or leveraging advanced techniques such as machine learning for anomaly detection or clustering to identify patterns beyond predefined bad smells. This approach could offer the benefits of both targeted analysis and comprehensive coverage while mitigating some of the drawbacks associated with each approach individually.

In deciding the best approach, it's essential to consider the specific goals, resources available, and the nature of the codebase being analyzed. Iterative refinement based on feedback and results obtained from initial analyses is also valuable in determining the most effective approach for a particular context.

**Results**: In this section, we present the gathered data and analyze the code metrics of the subject programs. We include graphs and tables illustrating the distribution of complexity, maintainability, and other relevant metrics across the analyzed fragments.

**Conclusion**: Through our analysis, we gain insights into the code quality of Android application fragments, highlighting areas of improvement and best practices for development. We discuss the implications of our findings and provide recommendations for developers to enhance the quality and maintainability of their fragment codebases.



1. Metrics Details For File 'src\main\java\com\zzg\mybatis\generator\controller\MainUIController.java'

--------------------------------------------------------------------------------------------

Parameter Value

========= =====

Project Directory C:\Users\Alekhya Changelpet\Desktop\Sem 1\Object Oriented Development\Group Assignmemt\Assign2\Programs\mybatis-generator-gui-master\

Project Name

Checkpoint Name AllFiles

File Name src\main\java\com\zzg\mybatis\generator\controller\MainUIController.java

Lines 528\*

Statements 455

Percent Branch Statements 10.1

Method Call Statements 233

Percent Lines with Comments 2.7

Classes and Interfaces 1

Methods per Class 26.00

Average Statements per Method 12.54

Line Number of Most Complex Method 121

Name of Most Complex Method MainUIController.displayTables()

Maximum Complexity 10\*

Line Number of Deepest Block 185

Maximum Block Depth 7

Average Block Depth 2.45

Average Complexity 3.77\*

--------------------------------------------------------------------------------------------

Most Complex Methods in 4 Class(es): Complexity, Statements, Max Depth, Calls

?(instance of Task<Void>).call() 1\*, 2, 6, 1

initialize().configsLabel.setOnMouseClicked() 1\*, 3, 3, 3

initialize().connectionLabel.setOnMouseClicked() 1\*, 3, 3, 3

initialize().filterTreeBox.addEventHandler() 2\*, 4, 4, 4

initialize().leftDBTree.setCellFactory(() 7\*, 24, 7, 15

initialize().useExample.setOnMouseClicked() 3\*, 4, 4, 3

leftDBTree.setCellFactory(().cell.addEventHandler() 7\*, 22, 7, 15

MainUIController.checkDirs() 9\*, 25, 7, 12

MainUIController.chooseProjectFolder() 2\*, 4, 3, 2

MainUIController.displayTables() 10\*, 39, 5, 28

MainUIController.generateCode() 8\*, 9, 5, 5

MainUIController.getGeneratorConfigFromUI() 1\*, 28, 2, 26

MainUIController.initialize() 10\*, 28, 7, 18

MainUIController.loadLeftDBTree() 3\*, 16, 4, 13

MainUIController.openTableColumnCustomizationPage() 4\*, 14, 4, 12

MainUIController.openTargetFolder() 1\*, 5, 3, 4

MainUIController.saveGeneratorConfig() 4\*, 18, 4, 14

MainUIController.setColumnOverrides() 1\*, 1, 2, 0

MainUIController.setGeneratorConfigIntoUI() 2\*, 27, 3, 28

MainUIController.setIgnoredColumns() 1\*, 1, 2, 0

MainUIController.setTooltip() 1\*, 10, 2, 10

MainUIController.validateConfig() 4\*, 8, 3, 8

--------------------------------------------------------------------------------------------

Block Depth Statements

0 42

1 87

2 145

3 69

4 58

5 29

6 21

7 4

8 0

9+ 0

--------------------------------------------------------------------------------------------

**Title**: **Analyzing Code Complexity and Quality Metrics of Java Files Using GQM Approach**

**Abstract**: This paper presents an analysis of code complexity and quality metrics of Java files utilizing the Goal-Question-Metric (GQM) approach. We investigate a dataset comprising Java files from various projects to assess their structural attributes and measure their adherence to coding standards. The study aims to provide insights into software maintainability and potential areas for improvement.

**Introduction**: Software quality assessment is crucial for ensuring maintainability, scalability, and reliability of software systems. Code complexity metrics offer valuable insights into the structural characteristics of source code, aiding developers in identifying potential areas for optimization and refactoring. In this study, we employ the GQM approach to establish clear objectives, formulate relevant questions, and define metrics for evaluating Java files' complexity and quality.

**Objectives, Questions, and Metrics:**

Our objectives revolve around assessing the complexity and quality of Java files to enhance software maintainability and development efficiency. We formulate the following questions:

1. What is the overall complexity of the Java files?

2. What is the distribution of complexity within the Java files?

3. How do the Java files adhere to coding standards and best practices?

**Subject Programs Description** : The dataset comprises Java files from diverse projects, including MainUIController.java. Each program serves specific functionalities within its respective project. MainUIController.java, for instance, serves as a controller class in a MyBatis generator project, managing user interface interactions and database operations.

**Tools Description:** For this analysis, we utilized a code analysis tool capable of extracting various metrics from Java files. The tool provides insights into code complexity, including cyclomatic complexity, method call statements, and block depth. Additionally, it offers metrics related to code quality, such as the percentage of lines with comments and class-to-method ratios. The tool's capabilities align with our objectives of assessing code complexity and quality.

**Approaches to Bad Smells Analysis** :

Comparing metrics for classes exhibiting bad smells against classes that don't is a common approach in software engineering and code quality analysis. However, there are different strategies you can employ:

**1. Comparing Bad Smell Classes vs. Clean Classes:**

- **Pros**: This approach directly identifies differences between problematic and clean code, aiding in understanding the impact of bad smells on metrics.

- **Cons**: It requires defining what constitutes a "bad smell" and may overlook subtler issues in supposedly clean code.

**2. Analyzing Metrics Across All Classes:**

- **Pros**: This approach provides a holistic view of code quality, allowing identification of outliers and trends across the entire codebase.

- **Cons**: It may obscure the specific impact of bad smells since metrics can be influenced by various factors beyond just the presence of bad smells.

**3. Developing a Hybrid Approach:**

- **Pros**: Combines the strengths of both approaches, providing detailed insights into bad smells' impact while also considering overall code quality trends.

- **Cons**: Requires careful consideration of how to balance the analysis and interpret results effectively.

When choosing an approach, consider the research goals, the context of the codebase, and the specific bad smells being investigated. Additionally, be mindful of the limitations of each approach, such as potential biases introduced by manual identification of bad smells or the sensitivity of metrics to codebase size and complexity.

**Results**: The analysis reveals significant insights into the complexity and quality metrics of the Java files. MainUIController.java, for instance, exhibits a high method-to-class ratio, indicating potential concerns regarding class cohesion. The distribution of block depth within the file highlights areas of code with varying levels of complexity. Moreover, the percentage of lines with comments suggests the level of documentation within the codebase.

**Conclusion**: Through the GQM approach, we effectively evaluated the complexity and quality metrics of Java files, providing valuable insights for software maintenance and enhancement. The findings underscore the importance of adhering to coding standards and refactoring complex code segments to improve overall software maintainability and development efficiency. Future research may explore automated refactoring techniques based on the identified metrics to streamline code optimization processes.

