International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

# Tool-Supported UML Analysis for Early Detection of Software Design Flaws

**Sweta Singh Patel*[1] & Arpana Bharani[2]**

**Submitted:**02/09/2024 **Accepted:**15/10/2024 **Published:**25/10/2024

**Abstract:** This research presents a UML-based approach for automated software architecture analysis aimed at improving design quality and maintainability in software systems. The proposed methodology integrates UML modeling, XMI-based data extraction, and Java-based metric computation to identify architectural weaknesses at an early stage. A case study on a Student Record System was conducted, where key UML diagrams—Component, Deployment, Class, and Use Case—were modeled using Enterprise Architect. A custom Java tool was developed to parse XMI files, extract structural information, and compute object-oriented metrics such as Lines of Code (LOC), Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Lack of Cohesion in Methods (LCOM), and Response for a Class (RFC). The metric analysis revealed concise, low-complexity, and well-organized classes, demonstrating the effectiveness of the approach in supporting maintainable and scalable software architecture. The tool's graphical interface allows users to visualize UML models, load XMI files, and perform automated architectural evaluation, making it accessible for both academic and professional applications. Overall, this study illustrates that automated UML-based metric evaluation provides a practical and efficient means to assess and enhance software design quality.

***Keywords***: *UML, XMI Parsing, Software Architecture, Automated Analysis, Design Flaws, Software Metrics, Enterprise Architect, Java tool.*

**Introduction:** Software architecture represents the high-level structure of a software system, defining the components, their interactions, and overall design decisions. A well-designed architecture ensures system reliability, maintainability, and scalability. However, as software systems become larger and more complex, manual analysis and detection of design flaws become time-consuming and prone to human error. Even small architectural defects can lead to significant performance degradation, poor maintainability, and higher development costs in later stages.

To overcome these challenges, the use of Unified Modeling Language (UML) and XML Metadata Interchange (XMI) has become a standard practice in model-driven software engineering. UML provides a graphical representation of system architecture, while XMI Serves as a machine-readable format that allows automatic data exchange between modeling tools and analysis systems.

This research focus on developing an automated approach to detect design flaws in software architecture using UML and XMI parsing. The proposed approach extracts architectural information from UML models and computes object-oriented design metrics such as weighted methods per class (WMC), Depth of inheritance Tree (DIT), Lack of cohesion of methods (LCOM), Number of children (NOC), and Response for a class (RFC). These metrics are then compared with defined threshold values to identify design anomalies, such as excessive complexity, deep inheritance, low cohesion, and high coupling.

Automation in the detection process minimizes human involvement and ensures consistent and repeatable results. The implementation of the proposed tool in Java and its integration with Enterprise Architect enables efficient parsing of XMI files and generation of detailed analytical reports. This study aims to enhance architectural evaluation accuracy, reduce maintenance efforts, and provide architects with actionable insights to improve software quality.

* [1]*Research scholar at Dr APJ Abdul Kalam University Indore, Madhya Pradesh*
[2]*Assistant professor at Dr APJ Abdul Kalam University Indore, Madhya Pradesh*
*Crossholding author address-Sweta.patel752@gmail.com*

**Literature Review:** Software architecture analysis has been a major area of research in software engineering, as it directly influences system quality and maintainability. Various researchers have proposed different approaches to analyze and evaluate software designs using UML models and metrics-based evaluations.

Several studies emphasize the importance of UML models for visualizing and understanding software structure. Booch, Rumbaugh, and Jacobson (1999) introduced UML as a standardized modeling language to represent software systems at multiple abstraction levels. Since then, UML has been widely adopted for modeling architectural components and their relationships.

Researchers such as Chidamber and Kemerer (1994) developed a suite of object-oriented metrics (CK Metrics), including WMC, DIT, NOC, LCOM, and RFC, to measure software quality attributes like complexity, cohesion, and coupling. These metrics became the foundation for automated evaluation of design quality.

Further studies integrated UML model with automated tools to reduce manual analysis. France and Rumpe (2007) discussed Model-Driven Engineering (MDE), which allows automated transformations and analysis using modeling languages like UML. Similarly, tools such as Enterprise Architect, MagicDraw, and StarUML, support exporting models in XMI format, enabling interoperability and automated data exchange.

In recent years, researchers have explored XMI-based parsing techniques to extract model information automatically. For example, Mishra et al. (2017) proposed a Java-based parser to analyze UML models for detecting design flaws through metrics comparison. However, most of these tools were either limited to specific metrics or lacked integration with complete architectural analysis.

The present research improves upon previous studies by designing a unified tool that automatically parses XMI files, calculates multiple metrics, and evaluates software architecture based on threshold values. This approach not only minimizes manual effort but also ensures consistency, scalability and accuracy in detecting architectural design flaws.

**Research Methodology:** The proposed research aims to develop an automated approach for detecting design flaws in software architecture using Unified Modeling Language (UML) and XML Metadata Interchange (XMI). The methodology integrates model-based design, automated metric extraction, and empirical analysis to identify architectural weaknesses during the design phase, thereby improving software quality and maintainability.

1. **Research Approach:** The proposed research follows a model-based analysis combined with tool implementation. The overall process consists of the following stages:

1. **Model Creation:** Develop UML models (Class Diagram) to represent the system architecture.
2. **XMI Export:** Export the UML models to XMI format using modeling tools such as enterprise Architect.
3. **Tool Development:** Design and implement a Java-based UML Analysis tool capable of parsing XMI files.
4. **Metric Extraction:** Extract important software design metrics such as
- Lines of Code (LOC)
- Weighted Methods per Class (WMC)
- Depth of Inheritance Tree (DIT)
- Number of Children (NOC)
- Lack of Cohesion in Methods (LCOM)
- Response for Class (RFC)
5. **Threshold Comparison**: Compare extracted metric values against standard threshold limits to identify design flaws and architectural risks.
6. **Report Generation**: Generate metric reports and graphical representations to visualize complexity and flaw detection results.

2. **Software Metrics via Class Diagram**
- A **Class Diagram** was modeled to represent the internal structure of the system, including its classes and relationships.
- The Class Diagram was then exported as an XMI file for further processing.
- A **custom Java-based Swing GUI tool** was developed to parse the XMI file and automatically extract key software architecture metrics.

These metrics serve as quantitative indicators for identifying potential design flaws and assessing the architectural quality of the software system.

3. **Tool Design and Implementation:** The proposed tool was developed using Java and XML parsing techniques. The process involves the following steps:
    - **Input:** XMI file generated from UML diagrams
    - **Process:**
        - Parse XMI elements (classes, attributes, operation, relationships)
        - Extract metrics automatically
        - Evaluate results against predefined threshold values

The tool assists in automated detection of structural design issues such as excessive complexity, poor cohesion, and deep inheritance, which are strong indicators of potential flaws.

4. **Empirical Analysis:** To evaluate the effectiveness of the proposed approach, multiple UML models representing different software systems were analyzed using the developed tool.Each model was processed to compute design metrics, and the results were compared with their respective threshold values.

**Table - Threshold Values Used for Evaluation**

| Metric | Threshold value | Description |
|---|---|---|
| LOC | >500 | Large class size may reduce maintainability |
| WMC | >20 | More methods may indicate high complexity |
| DIT | >5 | Deeper inheritance may lead to design issue |
| NOC | >10 | More children could signal improper abstraction |
| LCOM | >0.8 | High value indicates poor cohesion |
| RFC | >50 | High number of methods/message affects testing |

The comparison of calculated metrics with these thresholds helped identify architectural risks, complexity, and maintainability concerns.

**UML-Based Design and Implementation:** Unified Modeling Language (UML) class diagrams were created using **Enterprise Architect (version 17.1)** to design and visualize the **Student Record System.**

**Class Diagram –** To model the static structure of the Student Record System, a Class Diagram was developed using Enterprise Architect (EA). The Class Diagram plays a critical role in object-oriented analysis and design. It shows the system's classes, their attributes, methods, and relationships such as inheritance, association, and aggregation.

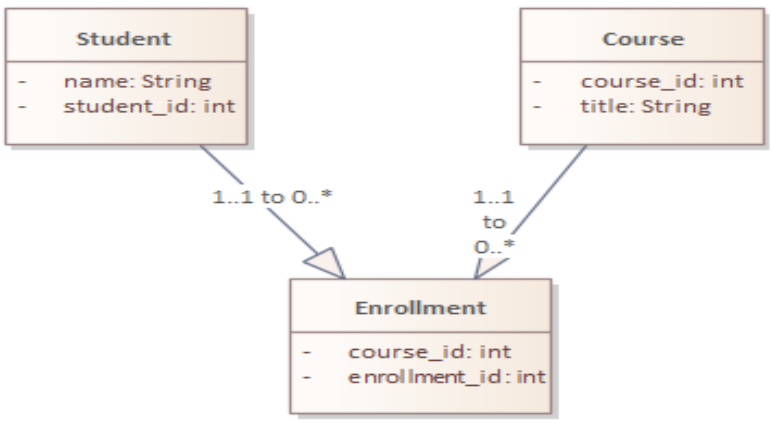


**Figure 1: Internal Structure of the Student Record System**

This diagram provides a blueprint of the internal structure of the software, making it ideal for metric – based analysis of software architecture quality.

**Classes in the Diagram:** The Class Diagram of the Student Record System includes the following key classes:

1. **Student**
   Attributes: studentID, name
2. **Course**
   Attributes: courseID, courseName
3. **Enrollment**
   Attributes: enrollmentID, date, status

**Relationships:**

- A Student can enroll in multiple Courses (association via Enrollment).
- The Enrollment class acts as a link between Student and Course (aggregation).
- Generalization, association, and navigable arrows are used where applicable.

**Role in Metric Analysis:**

The Class Diagram is the core diagram for evaluating software quality using object-oriented metrics. The Java tool developed for this research parses the XMI file generated from this diagram and extracts various metrics. The values of these metrics help in analyzing the maintainability, complexity, cohesion, and design quality of the software architecture.

**Description:** The Class Diagram illustrates the associations between the Student and Enrollment classes (one-to-many), as well as between the Enrollment and Course classes. This structure was subsequently exported in XMI format for metric analysis. Designed using Enterprise Architect, the diagram serves as the foundation for object-level analysis, enabling the application of software metrics such as LOC, WMC, DIT, NOC, RFC, and LCOM.

These models were exported in XMI (XML Metadata Interchange) format for further processing using the Java-based tool.

**Metric Evaluation Approach:** To evaluate software quality, the following object-oriented metrics were calculated:

- **Lines of Code (LOC):** Measures the size of the code.
- **Weighted Methods per Class (WMC):** Measures class complexity.
- **Depth of Inheritance Tree (DIT):** Indicates inheritance levels.
- **Number of Children (NOC):** Counts subclasses derived from a class.
- **Lack of Cohesion in Methods (LCOM):** Evaluates method-level cohesion.
- **Response for Class (RFC):** Measures the number of methods that can be executed in response to a message received by an object.

Each metric was analyzed using **standard threshold values** from software engineering guidelines to determine whether design elements met acceptable quality standards.

**Generation and Use of XMI Files:** Class diagrams were created in Enterprise Architect 17.1 and exported in XMI (XML Meta Data Interchange) format. The exported XMI files were used as input to the custom Java-based parser: These files included:

1. **Class Diagram XMI Files**


**Files 1: – Class Diagram from Exporting UML to XMI**

**Java-Based Tool Development and Metric Calculation:** A custom Java-based tool was developed using IntelliJ IDEA to parse XMI files and automatically extract software metrics from UML models. The tool facilitates automated metric evaluation and visualization, enabling effective analysis of software architecture quality.

**Tool Architecture:** The tool consists of the following key components:

- **XMI Reader Module:** Parses XMI files and identifies UML elements such as classes, attributes, and relationships.
- **Metric Calculator Module:** Analyzes the Class Diagram extracted from the XMI file generated by Enterprise Architect (version 17.1) to compute software metrics.
- **Output Generator Module:** Displays and visualizes the calculated results through an interactive Java Swing GUI.

This modular architecture ensures separation of concerns, simplifies maintenance, and enables automated metric computation directly from UML design artifacts.

**Program 1: Student.java**

```
Public class Student {
  int studentId;
  String name;
  //Constructor
Public Student (int studentId, String name) {
   this.studentId = studentId;
   this.name = name;
}
 // Display Method
Public void display(){
System.out.println("StudentId: " + studentId);
System.out.println("Name: " + name);
}
}
```

**Figure 1: Program Student.java**

**Program 2: Course.java**

```
Public class Course {
  int courseId;
  String courseName;
   //Constructor
Public Student (int courseId, String courseName) {
   this.courseId = courseId;
   this.courseName = courseName;
}
 // Display Method
Public void display(){
System.out.println("CourseId: " + courseId);
System.out.println("CourseName: " + courseName);
}
}
```

**Figure 2: Program Course.java**

**Program 3: Enrollment.java**

```
Public class Enrollment {
Student student;
Course course;
String enrollmentDate;
// Constructor
Public Enrollment(Student student, Course course,String enrollmentDate){
this.student = student;

this.course = course;
this.enrollmentDate = enrollmentDate;
}
// Display Method
Public void display(){
System.out.println("Enrollment Details:");
Student.display();
Course.display();
System.out.println("EnrollmentDate:" + emrollmentDate);
}
}
```

**Figure 3: Program Enrollment.java**

**Program: 4 Main.java**

```
Public class Main {
Public static void main(String[] args){
Student student1 = new Student(1,"Sweta Singh");
Course course1 = new Course(101,"Computer Science");
Enrolment enrollment1 = new Enrolment(student1,course1,"11 June 2025");
Enrollment1.display();
}
}
```

**Figure 4: Program Main.java**

**CODING IN JAVA**

All system classes were implemented in Java**,** with each class containing relevant attributes and methods to store and display data. The Main.java class was responsible for creating objects of the Student**,** Course, and Enrollment classes and invoking their methods to simulate the system's real-world behavior.

**Compilation and Execution**

- The program was compiled in IntelliJ IDEA by selecting Build → Build Project from the menu.
- After successful compilation, the program was executed by running Main.java using the Run button.
- Output was displayed in the IntelliJ console**.**

**OUTPUT:** The program successfully displayed **student names, courses enrolled, and enrollment IDs**, confirming correct interaction between the three classes. A sample output is shown below:


**Figure: Program Result Output**

**Architecture Analysis**

To evaluate the designed software architecture of the Student Record System**,** a set of object-oriented software metrics was applied. These metrics provide a quantitative basis for assessing the quality, complexity, and maintainability of the architecture. The system, modeled using UML, includes three primary classes: Student, Course, and Enrollment.

**Software Metric Calculation:** The following **object-oriented metrics** were calculated after implementing the system classes:

- **LOC (Lines of Code):** Measures class size and estimates development effort and complexity.
- **WMC (Weighted Methods per Class):** Sum of method complexities within a class. In this study, each method is assigned a complexity value of 1. WMC estimates effort required to develop, understand, and maintain the class.
- **DIT (Depth of Inheritance Tree):** Maximum length from a class to the root of the inheritance hierarchy. Higher values indicate increased complexity. *(No inheritance is used; DIT = 0 for all classes.)*
- **RFC (Response for a Class):** Total number of methods that can be invoked in response to a message received by the class. Higher RFC values indicate greater complexity, affecting understand ability, testing, and maintenance.
- **NOC (Number of Children):** Number of immediate subclasses derived from a class. A higher NOC can indicate reuse but may increase maintenance complexity.
- **LCOM (Lack of Cohesion in Methods):** Measures method dissimilarity within a class. Higher LCOM values suggest low cohesion, indicating potential design issues and reduced maintainability.

These metrics help identify potential risks**,** maintainability issues**,** and design weaknesses early in the development lifecycle, supporting informed architectural decisions.

**Software Metrics Table:**

**Table 3.4: Software Metrics Table**

| Metric | Student | Course | Enrolment |
|---|---|---|---|
| LOC(Line of code) | 10 | 10 | 13 |
| WMC(Weighted Methods per class) | 1 | 1 | 3 |
| DIT(Depth of Inheritance tree) | 0 | 0 | 0 |
| NOC(Number of children | 0 | 0 | 0 |
| RFC(Response for a Class) | 1 | 1 | 3 |
| LCOM(Lack of Cohesion) | Low | Low | Low |

**Summary of Metrics Evaluation:** The metrics extracted from the **Class Diagram** provide valuable insights into the internal structure and quality of the software system:

- **Lines of Code (LOC):** The Student, Course, and Enrollment classes ranged from 10 to 13 **LOC**, indicating concise implementations that enhance readability, simplify debugging, and improve maintainability. Small classes generally follow the Single Responsibility Principle and are less prone to errors.
- **Weighted Methods per Class (WMC):** Values of 1, 1, and 3 reflect minimal internal complexity, reducing cognitive load for developers and supporting future scalability and modifications.
- **Depth of Inheritance Tree (DIT):** All classes have a DIT of **0**, indicating a flat hierarchy without inheritance. While deeper inheritance can support reuse, a flat structure simplifies design and avoids potential complexity from polymorphism.

These results suggest that the system's architecture is well-structured, maintainable, and easy to understand, with classes designed to minimize complexity and maximize clarity.

**Conclusion:** This research demonstrates the effectiveness of a **UML-based approach for automated software architecture analysis**. By integrating **UML modeling, XMI parsing, and Java-based metric computation,** the proposed methodology enables early detection of design flaws and provides quantitative insights into the quality, complexity, and maintainability of software systems. The case study on the **Student Record System** showed that the developed tool accurately extracts key object-oriented metrics—such as LOC, WMC, DIT, NOC, LCOM, and RFC—from UML class diagrams, revealing concise, low-complexity, and well-structured classes. The **Java Swing-based graphical interface** allows users to visualize models, perform evaluations, and interprets results efficiently, making the tool practical for both academic and professional use. Overall, this study confirms that **automated metric extraction from UML diagrams** is a valuable approach for improving software design quality, supporting informed architectural decisions, and enhancing maintainability and scalability in the software development lifecycle.

**References**


[1] Gill N S., Grover P. S., "Software Size Prediction Before Coding," ACM SIGSOFT Software Engineering Notes, Vol. 29, Issue 5, Page 1-4, 2004.

[2] Jacobson I., "Object-Oriented Software Engineering. A Use Case Driven Approach", Addison-Wesley 1993.

[3] Karner G., "Metrics for Objectory", Diploma thesis, University of Linköping, Sweden. No. LiTH-IDA-Ex-9344:21, December 1993.

[4] Kim S., Lively W., Simmons D., "An Effort Estimation by UML points in the early stage of software development", proceedings of the 2006 international conference on software engineering research & practice, p 415-421, June, 2006.

[5] Kusumoto S., Matukawa F., Inoue K., Hanabusa S., and Maegawa Y., "Estimating Effort by Use Case Points: Method, Tool and Case Study," Proceedings of the 10th International Symposium on Software Metrics METRICS'04, (September14-16, 2004), pp. 292 – 299.

[6] Mahmood, S., Lai, R., Kim, Y.S., Kim, J.H., Park, S.C. and Oh, H.S., "A survey of component based system quality assurance and assessment", Information and Software Technology 702 47, pp 693–707, 2005. (DOI: 10.1016/j.infsof.2005.03.007)

[7] Massimo C., Giuseppe S., "Fast & Serious: a UML based metric for effort estimation", 6th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering, J, , Rome, Italy , Page 166-170, 2002.

[8] Mohagheghi P., Anda B., Conradi R., "Effort estimation of Use Cases for incremental large-scale software development", International Conference on Software Engineering (ICSE), 2005, pp.

303-331. (DOI: 10.1109/ICSE.2005.1553573 )

[9] Mili A, Chmiel S F, Gottumukkala R , Zhang L, "An integrated cost model for software reuse", In Proceedings of the 22nd international conference on Software engineering, 2000, pp. 157–166. (DOI: 10.1109/ICSE.2000.870407)

[10] Minkiewicz A. F., "The real costs of COTS", In Proceedings of IEEE Aerospace Conference, (USA, March, 2001), pp. 2863–2869.

[11] Narasimhan V. L., Hendradjaya B., "Theoretical considerations for software component metrics", Proceedings of World Academy of Science, Engineering and Technology, Volume 10, Page 165-170, 2005.