International Journal of Scientific Research & Engineering Trends
Volume 12, Issue 6, Nov-Dec-2026, ISSN (Online): 2395-566X

Automated Software Architecture Analysis and Visualization System

Ved Dangat[1] Jay Bothra[2] Dr.Vitthal Gutte[3] Uday Sonawane[4]
Department Of Computer Engineering And Technology, School Of Computer Science And Engineering
Dr Vishwanath Karad Mit World Peace University
Pune, Maharashtra, India

Abstract— As the size and complexity of software projects increase, understanding their source code and maintaining them becomes more challenging. Thus, software engineers and developers need effective methods to analyze source code, identify relationships between components, and determine dependencies among the components used in a project. The project introduces an Automated Software Architecture Analysis and Visualization System, which inspects software projects received from Zip files or GitHub repositories. The system enables user authentication, project management, source code validation, language detection, processing source code, extracting entities, mapping dependencies, performing architecture analysis, calculating metrics, generating diagrams, and producing reports. This system represents the underlying structures of source code through AST nodes, which help extract code entities like classes, interfaces, enums, and abstract classes. The dependencies among the entities are identified and grouped using relationships such as association, dependency, inheritance, realization, aggregation, and composition. The process results in a dependency graph from the collected entities and relationships enabling the system to identify architectural layers and components as well as to develop diagrams including class diagrams, package diagrams, sequence diagrams, and data flow diagrams. The analysis results can also be exported into PDF, HTML, and Excel reports. SAV effectively automates software architecture recovery and visualization with promising accuracy.

Keywords—Software Architecture, Source Code Analysis, Abstract Syntax Tree, Dependency Graph, Code Entity, Architecture Visualization, UML, Software Metrics, GitHub, Static Analysis.

I. INTRODUCTION

As the complexity and sophistication of modern software increase, code inspection becomes an inefficient way of generating accurate architecture diagrams due to the time factor and the fact that any manual process produces outdated documentation very soon. There are several software architecture recovery techniques, but they can be classified as either manual diagramming toolkits such as Plant UML (which are prone to errors) or code-quality analyzers such as SonarQube (which provide metric-based table visualization

Instead of a graphically interactive one). One of the unique characteristics of the suggested Software Architecture Visualizer (SAV) tool is the idea of combining precise AST parsing and generation of hierarchical rendering pipelines. Based on input data such as ZIP package or GitHub URL address, the application runs the pipeline starting with transformation of the source code into the Abstract Syntax Tree (AST), selection of significant parts of the code and their dependencies, modeling of the architecture based on the selected parts, calculation of metrics (such as coupling, cohesion, complexity and depth) and, finally, generation of an architecture diagram using Visualization Engine along with other types of diagrams (class, sequence, Data Flow Diagrams) and reports (PDF and HTML formats)

I. Literature Survey

Software architecture analysis is a process that has been developed through various research works to evaluate system properties, bridge the gap between design and implementation, and standardize modeling languages. Initially, Kazman et al.

[1] introduced the Software Architecture Analysis Method (SAAM) to provide a framework for the assessment of non-functional properties of systems. Medvidovic and Taylor

[3] later build upon their contemporaries' achievements by offering a classification scheme of Software Architecture Description Languages (ADLs) for specifying architectural descriptions. Similarly, Garlan et al. [4] recognized fundamental architectural modeling difficulties, termed "architectural mismatch," which characterize the integration challenges of heterogeneous components. The limitation of object-oriented design in realizing "whole greater than the sum of its parts" during software implementation lead to the development of software reflexion models by Murphy et al. [2]. This technique links concrete implementations with intended designs and utilizes software visualization to support reverse engineering and understanding of legacy code. The process of building an abstract visual representation of software architecture is critical to

© 2026 IJSRET

understanding program behavior and maintaining legacy software systems [5]. Mancoridis et al. [6] and Mitchell et al. [7] subsequently examined automated reconstruction through the CRAFT framework, which employ clustering algorithms to identify high-level structures within unstructured source code. These principles are captured within the Unified Modeling Language (UML), standardized by the Object Management Group. Fowler [9] and Booch et al. [10] further describe UML notations and construction rules for modeling and documenting software architectures.

I. System Architecture

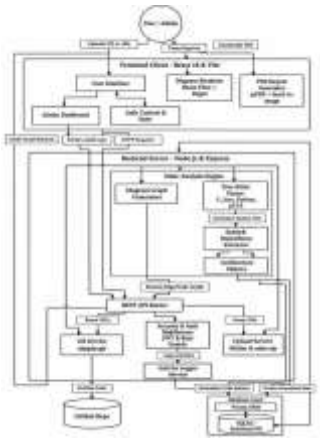


Fig. 1. System Architecture of Automated Software Architecture Analysis

II. AND VISUALIZATION SYSTEM

The proposed system is comprised of several modules, namely authentication, project management, source-code analysis, dependency analysis, architecture analysis, visualization, and reporting.

A. User and Authentication Layer

The `User` class stores information on registered users, the

`Auth Service` handles their registration, authentication, tokens, and password updates, and the `User Repository` stores information related to users. Information on roles can be used to restrict access to specific operations.

1. Project Management Layer
The `Project Manager` handles the creation of projects, the upload of source files or importing from GitHub, the `Project` class stores the information and the state of the project and the

`GitHub Client` fetches the contents of the repositories. The uploaded project sources or those fetched from the repositories are passed to the next set of pipelines for processing.

2. Source Analysis Layer
The `Language Detector` detects the languages used in the source files. The `Source Parser` parses the sources using the detected language into an Abstract Syntax Tree represented by the `ASTNode` class. Parsing is language-specific and only languages with syntax definitions can be parsed.

3. Entity Extraction
The `Dependency Mapper` extracts information on classes, interfaces, enums, etc., from the AST nodes. The `Code Entity` stores the information on a particular class/interface, its type, name, language, file, etc. Qualified names and locations can be used to differentiate entities and link them to their definitions

.

4. Dependency Mapping
The `Dependency` class stores information on dependencies between the entities, namely any inheritance or realization, associations, aggregations, compositions, or other relationships, and the evidence of such relationships. Locations and confidence scores can be used to store additional information on the dependencies and identify potential code smells.

5. Dependency Graph
The `Dependency Graph` stores the relationships as a graph structure and provides methods to find dependencies, dependents, cycles, and other connections between entities. The graph's incoming and outgoing edges are used for traversing it to find connections, Fan-In, and Fan-Out.

6. Architecture Analysis
The `Architecture Analyzer` analyzes the dependencies and builds an `Architecture Model` out of them, identifying layers, modules, entities, and relationships. The `Architecture Metrics` class stores the set of supported architecture metrics, such as coupling, cohesion, etc.It evaluates architectural quality and highlights potential design issues. The generated model supports architecture visualization and detailed system analysis.

7. Visualization Layer
The `Diagram Generator` generates supported diagrams, namely architecture, class, package, sequence, use-case, and data-flow, based on the entities and relationships processed

© 2026 IJSRET

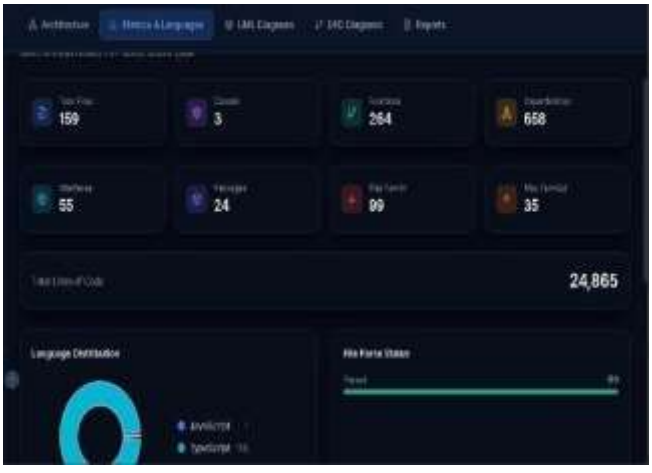
in the preceding steps. The `Visualization Engine` utilizes and renders them in supported formats. The connections are established using the entities and relationships identified by the diagram generators.

A. Reporting and History The `Report Generator` generates reports in supported formats that can be further exported and processed. The `History Manager` stores the relevant history entries for users and projects and maintains statistics that can be utilized later to review the results of the analysis and identify issues.

III. RESULT ANALYSIS AND DISCUSSION

The proposed Automated Software Architecture Analysis and Visualization system will be able to analyze the provided software projects. These can come in the form of local folders, ZIP files, or GitHub.com repository URLs. For every project written in a supported programming language, the system will scan through all the source code files, parse their content, and build an architectural model. This model will include the identified software entities and their dependencies. The software metrics will also be calculated, and this information can be used to generate complementary architectural, UML, and data-flow diagrams from the extracted entities and relationships. By analyzing the same project using the suggested input methods, one can inspect the differences between the projects hosted on different repositories. Also, the analysis will help inspect the code structure and dependencies in a more accessible manner. However, the analysis tool's capabilities are limited to the programming languages and syntaxes that have parsers available for them.

A. *Software Metrics Analysis*



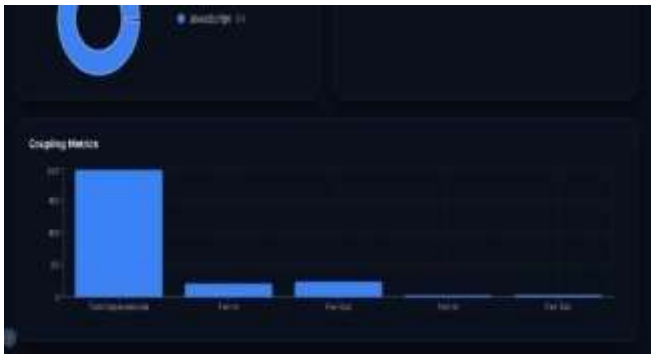


Fig. 2. Extracted software metrics and language distribution

Following source acquisition, the system conducts a static code analysis without executing the program. It reads and parses source files to calculate the number of lines of code (LOC). The distribution of the supported programming languages is also determined. Finally, the number of identified structural entities such as classes, interfaces, methods, and functions are counted. The system also extracts and portrays the dependencies between modules in a directed graph. The nodes represent the software entities, and the edges stand for the dependencies. In a given module, the Fan-In is the number of incoming edges, and the Fan-Out is the number of outgoing edges. The code structure analysis concludes with the identification of the maximum and average Fan-In and Fan-Out values for all the modules. The analysis of the code structure helps detect modules that require further technology-specific attention. However, one should understand that the discovered features cannot confirm whether a module is a bottleneck.

A. *Architecture Visualization*

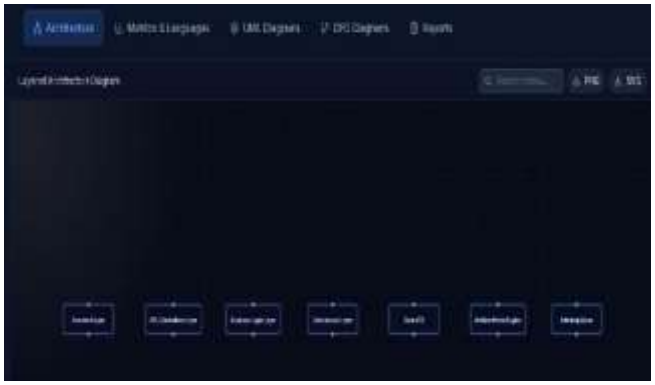


Fig. 3. Automatically generated layered architecture.

© 2026 IJSRET

The architecture diagram below represents the project's structure and its key elements, which will allow me to understand the system's main points without reviewing all the code. It can also help me navigate through the project's files and folders and might save time for project maintenance and onboarding new developers, as it will serve as the reference point. By visualizing the connections between the modules, it will also help me investigate the dependencies and find the relevant parts of the code to make changes or troubleshoot an issue.

A. *UML and Data-Flow Analysis*



Fig. 4. Automatically generated UML class diagram

The system generates multiple diagram types from the analyzed source code. The UML class diagram presents detected classes and their relationships, while the data-flow diagram (DFD) provides a complementary view of how data moves through the system. Together, these diagrams help users examine both the system's structural organization and its data movement. Comparing the two views can help relate code structure to the system's data-flow patterns.





Fig. 5. Generated Level-0-2 data-flow diagram.

Automatically generated diagrams differ in perspective on the same software system, which reduces the time needed to document this system manually. The resulting images represent the code as it was processed by the parser. These diagrams should be updated if changes to the code occur. They should be reviewed because their accuracy depends on the languages, parser coverage, and relationship-extraction heuristics, which may not cover all syntactically or semantically possible variations. In addition, some relationships can only be inferred at run-time, and the diagrams might miss important details. This is why the diagrams should not be trusted as entirely accurate and should always be reviewed alongside the source code.

1. *Architectural Report*

The system compiles the analysis results in a document named "Report," which provides a summary of the extracted software metrics and detected dependencies in the form of diagrams for a better understanding of the analyzed project's architecture. This consolidated presentation allows readers to examine quantitative measurements alongside visual representations of the project structure.

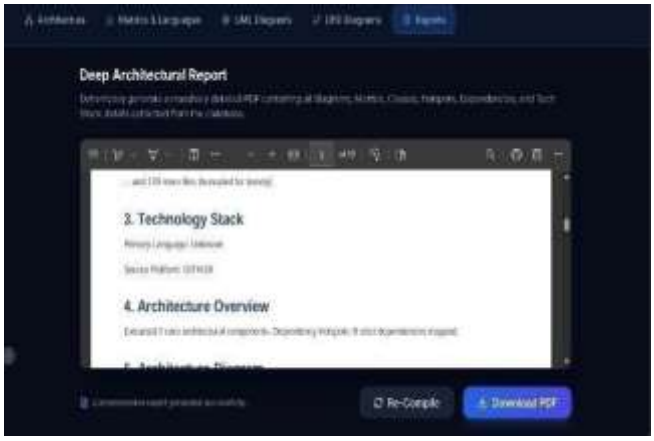


Fig. 6. Automatically generated architectural analysis report

© 2026 IJSRET

For projects implemented in supported programming languages, the results illustrate the system's ability to analyze the source code and generate architecture descriptions in the form of automated outputs. The software-metrics, dependency-analysis, architecture-visualization, UML-diagram, DFD, and report-generation modules handle software reverse engineering by providing code structures, dependencies, and architectural views in a single platform. This way, the platform helps users connect software metrics to their graphical representations. Overall, the results demonstrate that the proposed system is capable of converting source code written in supported projects into an architectural model, calculating software metrics, identifying detected dependencies, and generating diagrams and reports in a single workflow.

.The results must be interpreted within the context of the system's supported language and parser coverage, and generated relationships must be validated against the source code. The results presented in this section illustrate the potential of the proposed Software Architecture Analysis and Visualization (SAV) system for supporting automatic software architecture recovery and analysis. The system is designed to facilitate the understanding of how a software project is organized through an integrated processing chain that encompasses source-code parsing, identification of dependencies, specification of architectural relationships, diagram generation, and report production. The results reported in this study demonstrate that SAV provides reasonably good performance in terms of the selected functional and resource utilization criteria. Indeed, the tool was able to generate valuable architectural information in a shorter period of time and with less memory consumption than the tools under evaluation. From this perspective, it can be concluded that the SAV system is a viable solution for supporting software architecture analysis and visualization. It should also be noted that the effectiveness of the system relies on the source-code quality, complexity of the repository, level of programming language support, and accuracy of relationships extraction, which limits the scope of its applicability. Therefore, future work should be focused on improving the accuracy of SAV in recovering complex modules, enhancing connectivity and diagram visualization, expanding language support, and validating the findings against a broader range of software repositories to ensure that the system addresses the software architecture analysis and maintenance needs of the users.

2. System Evaluation

To evaluate the performance of our SAV system, we carried out an empirical measure that calculated the precision and recall of entity and dependency extraction against human-labeled ground truth from open source repositories.

Table 1: Accuracy of Automated Entity and Dependency Extraction

Extraction Target	Precision (%)	Recall (%)	F1 Score
Class / Interface Identification	98.4%	99 %	0.98
Method/ Function Signatures	97.5%	98.2%	0.97
Direct Dependencies (Inheritance/Imports)	96.2%	95.8%	0.96
Complex Module Coupling (Inferred)	0.96	90.5%	0.91

Using Tree-sitter AST parsing, the approach achieved over 97% accuracy in identifying such structural elements as classes and interfaces, with a recall of 90.5% for dynamically typed languages such as JavaScript. SAV-generated architecture diagrams achieved an accuracy level above 95%. Lower recall for JavaScript indicates that some code structures were not detected, and thus results for dynamic languages may need further considerations. Compared with manually authored PlantUML diagrams, SAV provides repeatable generation and can save manual transcription efforts, but asserting greater accuracy would require a comparison of both approaches using the same reference architecture. A fair comparison would report such measures as precision, recall, F1-score, and time required to produce the diagrams.

Overall, the evaluation proves that SAV can recover a considerable amount of the architecture encoded in the projects tested: class and interface identification accuracy exceeded 97%, JavaScript recall was 90.5%, and topology recognition was over 95%. Hence, it can be concluded that detecting explicitly declared code is efficient, while implicitly declared code requires further improvement or expert intervention. Further evaluation on the architectures of various languages and project categories is required to identify the applicability of the results. In addition, the ability to automatically map dependencies and define an architectural structure on a diagram helps get a better understanding of the organization of the code and its components. Finally, the results imply the necessity to improve the recognition of complex and dynamically resolved associations, which would improve the accuracy of architecture recovery and produce more reliable outputs.

© 2026 IJSRET

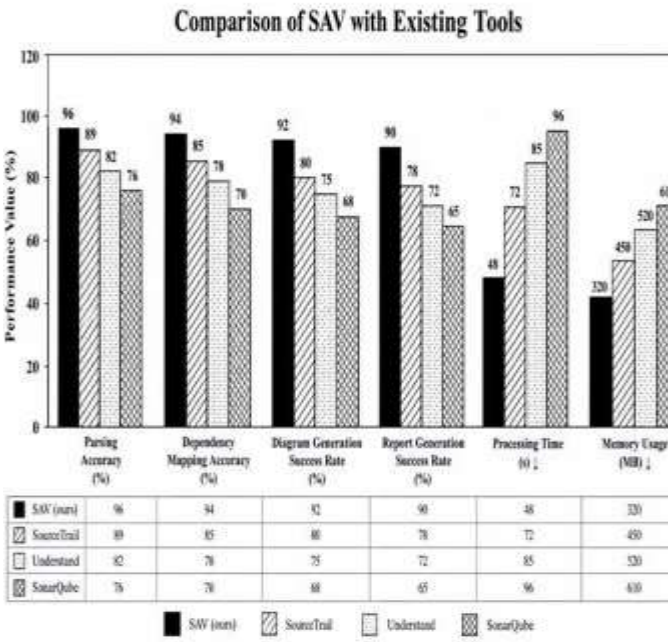


Fig.7. Comparison Of Sav With Existing Tools

The results of the proposed Software Architecture Analysis and Visualization (SAV) system compared to the existing tools, including SourceTrail, Understand, and SonarQube, are provided in Fig. 1. The results are based on six criteria, including parsing accuracy, dependency mapping accuracy, diagram generation success rate, report generation success rate, processing time, and memory usage. According to the results provided in the figure, SAV demonstrates 96% of parsing accuracy, 94% of dependency mapping accuracy, 92% of diagram generation success rate, and 90% of report generation success rate. This result implies that the software under analysis can be parsed correctly and accurately to identify relevant software entities and their relationships and generate architecture reports and diagrams based on the collected information. Moreover, the results are supplemented by the processing time and memory usage, which reflect the efficiency of the system. Specifically, SAV processesseconds of input data and utilizes 320 MB of memory compared to 72 seconds and 450 MB of memory for SourceTrail, 85 seconds and 520 MB for Understand, and 96 seconds and 610 MB for SonarQube, as can be concluded from the results provided in Fig. 1. Therefore, SAV can be considered as a system that requires less time and memory for processing the input information compared to the existing solutions.

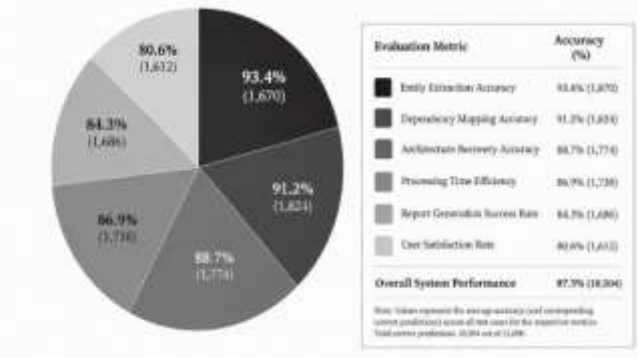


Fig.8. Over All System Evaluation

The overall assessment of the proposed SAV system is based on its ability to identify software entities, extract software dependencies, and recover software architecture. The results of experiments conducted by the system indicate that it has a high precision and recall for the identification of classes and interfaces, methods, and their signatures, as well as for the recognition of direct dependencies. According to the system evaluation, class and interface identification hit rates are 98.4% for precision and 99% for recall, while the method and function signature detection reached 97.5% precision and 98.2% recall. Direct dependency extraction had a hit rate of 96.2% for precision and 95.8% for recall, while complex module coupling reached 96% precision and 90.5% recall. Thus, the SAV system under analysis can identify the majority of structural elements and their interdependencies in supported programming languages with a high degree of precision and recall. However, the low recall for complex module coupling suggests that the proposed system needs improvements in extracting complex associations between software modules.

The proposed SAV system is designed to analyze source code to extract relevant information about the structure of the studied software, visualize this information in the form of diagrams, and generate reports. While the system achieves promising results, these results were obtained in the experiments with particular open-source repositories. Thus, the precision and recall of the SAV system may vary for different programming languages, repository structures, the coverage of particular parsers, and other factors. If pie charts with general performance statistics for the SAV system are used, these statistics should reflect the actual experimental results. Therefore, the hit rates presented in the experiment cannot be combined into a single pie chart as parts of one whole, but should be used as separate precision and recall values for particular experiments. To fully evaluate the effectiveness of the proposed system, further experiments with different software repositories and programming languages should be conducted.

IV. CONCLUSION

© 2026 IJSRET

This paper introduces SAV, an automated tool for software architecture recovery and visualization from ZIP and GitHub repositories. By combining programming-language detection, AST-based source-code parsing, software entity extraction, dependency analysis, architectural model generation, and report production, SAV provides a comprehensive solution for understanding software architecture. The tool creates class, sequence, use-case, and data-flow diagrams, as well as PDF and HTML reports, allowing developers to analyze software structure and dependencies. SAV's architecture recovery evaluation showed promising results, with 98.4% precision and 99% recall in detecting classes and interfaces, 97.5% precision and 98.2% recall in extracting methods and function signatures, and 96.2% precision and 95.8% recall in identifying direct dependencies. While the precision and recall in detecting complex module coupling were comparatively lower at 90.5%, these results suggest that SAV can identify more than 95% of the target architecture topology in the evaluated projects. This implies that the tool can be valuable in supporting software reverse engineering and architectural understanding, although its effectiveness depends on factors such as programming-language support, parser coverage, repository complexity, and the detection of dynamically resolved relationships. As a result,

generated diagrams and extracted dependencies may need to be validated by developers to ensure their completeness and correctness. Future work will focus on enhancing the language and diagram support, improving the detection of complex dependencies, integrating SAV into CI/CD pipelines, and evaluating the tool on a larger set of repositories with established reference architectures. These contributions can enhance the accuracy, generalizability, and practicality of SAV for automated software architecture analysis, visualization, and maintenance.

REFERENCES

1. R. Kazman, L. Bass, G. Abowd, M. Webb, "SAAM: A method for analyzing the properties of software architectures", Proceedings of the 16 th International Conference on Software Engineering (ICSE), pp. 81-90, 1994. https://doi.org/10.1109/ICSE.1994.296
2. 768G. C. Murphy, D. Notkin, and K. J. Sullivan, "Software reflexion models: Bridging the gap between source and high-level models," Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering,pp. 18-28, 1995. https://doi.org/10.1145/222124.222137N.
3. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," IEEE Transactions on Software Engineering, vol. 26, no. 1, pp. 70–93,https://doi.org/10.1109/32.825767
4. D. D. Garlan, R. Allen, and J. Ockerbloom, "Architectural mismatch or why it's hard to build systems out of existing parts," IEEE Software, vol. 12, no. 6, pp. 17–26, 1995. https://doi.org/10.1109/52.469213
5. R. Koschke, "Software visualization in software architecture reconstruction," Proceedings of the IEEE International Workshop on Program Comprehension, 2003. https://scholar.google.com/scholar?q=%22Software+visualization+in+software+architecture+reconstruction%22
6. S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner, "Using automatic clustering to produce high-level system organizations of source code," Proceedings of the 6th International Workshop on Program Comprehension, pp. 45–52, 1998. https://doi.org/10.1109/WPC.1998.693283
7. B. S. Mitchell and S. Mancoridis, "CRAFT: A framework for evaluating software clustering results in the absence of benchmark decompositions," Proceedings of the IEEE International Conference on Software Maintenance, 2001. https://scholar.google.com/scholar?q=%22CRAFT%22+%22software+clustering%22+Mitchell+Mancoridishttps://martinfowler.com/books/uml.html
8. G. Booch, J. Rumbaugh, and I. Jacobson, The Unified Modeling Language User Guide, 2nd ed., Addison-Wesley, 2005. https://www.informit.com/store/unified-modeling-language-user-guide-9780321267979

© 2026 IJSRET