**Project Title : “Integrating Reverse Engineering and Visualization Techniques for Effective Software Maintenance”**

**Group Student Name:**

24-93245-1 Chowdhury, Shah Fahim

24-93566-3 Lima , Rashida Begum

25-93648-1 Jahan, Nousheen

**MSc Program:** MSCS

**Course Title: Software Evolution and Maintenance**

**Supervisor Name:**

Dr. Mohammad Rabiul Islam

**Institution Name:** American International University-Bangladesh

**Submission Date:** 25 April 2025

# 1. Introduction

Software systems are always changing because of new business needs, technology changes, and operational needs. This ongoing process is referred to as software maintenance. Software maintenance is a vital activity to ensure the reliability, adaptability, and ongoing value of a software system. Maintenance is more than just fixing bugs; it also includes improvements, performance, and adaptation to changed environments[4].

As systems become larger and more complex, software maintenance is becoming more difficult. This is especially true with legacy systems which can involve outdated documentation, tightly coupled code, and loss of knowledge of the original design. In these situations, reverse engineering, and software visualization are needed.

Reverse engineering allows you to recover design, architecture, and behavior from code. This is often code at a low level that can be reverse engineered into abstract ideas. Software visualization is used to recover the understanding of a program, producing graphical representations of software structure, control flow, and dependencies.

Using these techniques together supports software maintainability, improves understanding of legacy systems, and supports Lehman's laws of software evolution, which emphasize change, complexity, and feedback. This project aims to investigate the relationship between reverse engineering and visualization to support program comprehension and propose a prototype framework for using reverse engineering and visualization together in the context of practical programming.

Software systems have continuous change as they are modified or transformed over time to respond to changing business needs; to accommodate upgrades to business technology; and/or to fulfill user's changing expectations and demands. While it is widely accepted that this ongoing and evolving process is a broad definition of software maintenance, it is also recognized that software maintenance is an important element of software quality, as it is often done not only to remove defects, but to improve function, performance, and the longevity of the system; however, software maintenance can be challenging when working with a large, complex, and poorly documented software system, especially if the original developers are unavailable[6].

The challenge to legacy software systems has legitimized the use of reverse engineering and software visualization as means to analyze, understand and maintain legacy software. More specifically, reverse engineering refers to the analytical process used to determine the component parts and relationships of an existing software system, often representing the original low-level code in high-level abstractions such as models, diagrams, or specifications. It is an important part of software maintainability, especially for legacy systems where the documentation is either out of date or missing altogether.

Software visualization techniques, on the other hand, refer to the graphical representations of software architecture, control flow, data flow, structure and dependencies etc. Each of these graphical representations provides valuable visual insights to significantly improve program understanding that allows software maintainers to make informed decisions and perform accurate modifications.

Using reverse engineering tools in conjunction with software visualization techniques is an very robust mechanism for software maintainers and provides a solid overview to better understand code structure and system behavior, in addition to hastening the process of finding and fixing bugs. Furthermore, this integrated approach compliments the laws of software evolution that inherently suggest that systems will continue to change while also getting more complex, requiring software maintainers to engage in continuous adaptation.

This project investigates how the combination of reverse engineering and visualization techniques can increase the understanding of programs, help with maintenance, and satisfy the principles of software evolution theories. It also includes design of a prototype tool that employs the combined effort in a real system.

# 2. Literature Review

### **2.1 Lehman’s Laws of Software Evolution**

M. M. Lehman proposed eight empirical laws of software evolution:

* Law 1 – Continuing Change: A system must continue to evolve, or it is going to lose utility.
* Law 2 – Increasing Complexity: The complexity of a software system will only increase unless managed actively.
* Law 3 – Self-Regulation: Evolution is a process that is semiautomatic, measurable, and self-regulating.
* Law 4 – Conservation of Organizational Stability: The amount of development effort put into a system follows a relatively constant function over time.
* Law 5 – Conservation of Familiarity: Change will end only when a project stops because then there is no need to maintain familiarity.
* Law 6 – Continuing Growth: Functional capability can only grow; functional capability cannot remain static.
* Law 7 – Declining Quality: The quality of a system will decline if not proactively maintained.
* Law 8 – Feedback System: Evolution is cetroled by both internal and external feedback systems.

These laws lay out the rationale of why software maintenance is unavoidable and why we need tools like reverse engineering and visualization to manage system complexity [1].

#### 2.2 Key Maintenance Processes and Strategies

Software maintenance has four important types.

* Corrective Maintenance: Repairs for defects discovered.
* Adaptive Maintenance: Adjustments made to the software system so it is used with another software system.
* Perfective Maintenance: Improvements in performance and/or maintainability of the software system.
* Preventive Maintenance: Solutions for hidden problems that are waiting to create changes in the customers' use of the software system.

Empirical studies identify major strategies to support maintenance, including:

* Reverse Engineering: Recovering design knowledge and code semantics.
* Refactoring: Changing code structure without changing semantics.
* Configuration Management: Keeping track of version, changes, and history.
* Tool Support: There are static analyzers, profilers, visualizers, and testing frameworks that can help make most of the work easier and less error-prone [5].

Each strategy and process adds to the best practices for evolving systems long-term [3].

### **2.3 Discussion About Legacy Systems**

Legacy systems are software systems that have been in use for a long time, have great business purpose, but are built using obsolete technology and architecture. Examples include:

* Being written in obsolete programming language (e.g., COBOL, FORTRAN)
* Badly documented or no documentation
* Monolithic and tightly coupled
* High business value, high risk and cost to change

There are several challenges to maintaining legacy systems:

* New developers may become scarce in these obsolete technologies
* Legacy systems may not be compatible with modern platforms
* New features or fixes may be difficult to implement

Possible Modernization techniques include:

Technique Description

Encapsulation Wrapping legacy functions with APIs

Rehosting Moving it to modern infrastructure and using it without code changes

Refactoring Cleanup and modularize what already exists

Reengineering Build it again but use modern tools preserving business logic

Reverse engineering and visualization can help ease the transition between obsolete systems to modern maintenance practice by providing:

* Structure
* Architecture
* Easing decision making

## 2.4 Table of Literature Review

|  |  |  |  |
| --- | --- | --- | --- |
| SI No | Citation | Title & Author(s) | Summary of Contribution |
| 1 | [1] | Software Maintenance and Evolution: A Roadmap – M. Lehman (2000) | Presents foundational concepts of software evolution and Lehman’s laws. |
| 2 | [2] | A Survey of Reverse Engineering and Program Comprehension – Chikofsky & Cross (1990) | Defines reverse engineering taxonomy and its use in legacy systems. |
| 3 | [3] | Visualizing Software for Understanding and Analysis: A Survey – Diehl (2007) | Reviews visualization tools for software structure and behavior. |
| 4 | [4] | A Metrics Suite for Object-Oriented Design – Chidamber & Kemerer (1994) | Introduces design metrics for evaluating software complexity. |
| 5 | [5] | Architecture Recovery using Dependency Analysis – Kazman et al. (1998) | Explains architecture recovery via dependency analysis. |

# 3. Framework Development

This section describes the design of a prototype framework that combines reverse engineering and visualization using the tool called Graphviz. The aim of the framework is to facilitate software maintenance activities, specifically understanding, documenting, and analyzing legacy system, by providing visual representations of code structures and flows[8].

## 3.1 Reverse Engineering Support

Reverse engineering is a process of examining software to recover its design, architecture, and functional behavior. This is often important when dealing with a legacy system that has lost original documentation, and where understandable complexity has increased over time.

The proposed framework uses Graphviz as its core enabling technology for reverse engineering and allows the user to extract code-level relationships such as function calls, module dependencies and control flow and place those relationships in visual form. These relationships will be declared using a simple scripting language called DOT, through which Graphviz interprets to produce diagrams[2].

The greatest advantage of this technology is its language-agnostic capacity; if developers can extract relationships, they can reverse-engineer any language or system by a manual process of define the relationships between entities in the system, or by writing a simple script to extract the relationships in the code[7].

## 3.2 Visualization Capabilities

Visualization is a substantial facilitator in software comprehension, particularly for maintainers and engineers who are unfamiliar with the original codebase. The framework uses Graphviz to produce the following diagrams:

* Call graphs - Diagrams showing what functions call what functions in the codebase.
* Class or module diagrams - Diagrams showing relationships between classes or files, useful when working with object-focused designs or modular architecture[13].
* Control flow diagrams - Diagrams that represent high-level logical paths or sequences of execution.
* Data-flow diagrams - Diagrams showing how data passes between functions or modules that help you understand how inputs get transformed into outputs.

All these visualizations are generated automatically by Graphviz after encoding the relationships and connections in a .dot file.

Having this visual output makes it easier to identify bypasses, duplicate modules, and burdensome interdependencies that put the system at risk as it is altered over time[12].

## 3.3 open-source libraries or custom logic

Graphviz is an open-source graph visualization application that means the proposed framework is inexpensive and easy to implement. Graphviz offers:

* Cross platform, open-source means it's easy to access and has no cost for distribution,
* Command line utilities (i.e. dot, neato, fdp) for rendering various types of graphs,
* Access to many scripting languages (Python, Java, Shell, etc.) for automation,
* Multiple output formats (e.g. PNG, SVG, PDF, interactive web visualizations).

The framework can be implemented manually by creating 'DOT' files, or via automated parsers or other transformation scripts. For example, it is possible to write a Python script to scan all .py files, look for function definitions, and generate a DOT file that describes their calling relationships[9].

This type of modularity contributes to the agility of the current framework for small- to mid-sized projects, and provides a necessary foundation for future more complex tooling[10].

## 3.4 Workflow of proposed Framework

The process adheres to a simple and repeatable workflow:

1. Extract relations from source code, either manually or by using automated scripts.
2. Declare these relations in a corresponding .dot file in DOT language.
3. Process the .dot file in Graphviz to generate a corresponding visual diagram.
4. Use the diagrams to analyze, document, and understand the system for maintaining the system[11].

This workflow helps both technical and non-technical stakeholders control structure and behavior, making it a useful addition to software maintenance activities.

## 4. Implementation

### 4.1 Selected Legacy Codebase:

1. We have chosen open-source “Simple Inventory Management System” developed in Python from GitHub. The codebase is a legacy-like system, meaning it was not well documented and mostly procedural.
2. 4.2 Application of Framework:
3. The framework prototype was executed as explained below:
4. Code Relationship Extraction:
5. • Parsed Python scripts for detecting function definitions and their invocations.
6. • Refactored inter-module dependencies out of the kernel.
7. DOT File Creation:
8. • A script was developed to generate . dot data out of the model data.
9. Visualization with Graphviz:
10. • Produced call graphs, module diagrams and control flow diagrams.
11. • Exported results as PNG and SVG files.
12. Example DOT Snippet:
13. digraph CallGraph {
14. main -> add\_item;
15. main -> update\_stock;
16. update\_stock -> validate\_item;
17. }
18. Outputs:
19. callgraph. function call hierarchy gnome-png subversion-plugin print-function-calls (Computer Science): A plugin for the gnome-png subversion-plugin which allows printing of a function call hierarchy.
20. module\_diagram. svg showing module interaction.

## 4.3 Implementation Script for Call Graph Extraction

To automate the process of function relationships extraction from the legacy Python codebase, we created a custom script by means of the Python’s own ast library. The code does the following:

• Parses each . py source file at the provided directory climatebne -d path-to-repo The following example command runs tests on all files in a directory python -m climatebne -d dir where dir contains the `.

• Recognizes function definitions and invocations.

• Gathers calling relationships.

• Outputs a . dot file ready for rendering with Graphviz.

import ast  
import os  
  
class CallGraphVisitor(ast.NodeVisitor):  
 def \_\_init\_\_(self):  
 self.calls = []  
 self.current\_func = None  
  
 def visit\_FunctionDef(self, node):  
 prev\_func = self.current\_func  
 self.current\_func = node.name  
 self.generic\_visit(node)  
 self.current\_func = prev\_func  
  
 def visit\_Call(self, node):  
 if isinstance(node.func, ast.Name):  
 called\_func = node.func.id  
 if self.current\_func:  
 self.calls.append((self.current\_func, called\_func))  
 self.generic\_visit(node)  
  
def extract\_calls\_from\_file(filepath):  
 with open(filepath, "r") as file:  
 tree = ast.parse(file.read(), filename=filepath)  
 visitor = CallGraphVisitor()  
 visitor.visit(tree)  
 return visitor.calls  
  
def generate\_dot(call\_edges, output\_file="callgraph.dot"):  
 with open(output\_file, "w") as f:  
 f.write("digraph CallGraph {\n")  
 for caller, callee in call\_edges:  
 f.write(f' {caller} -> {callee};\n')  
 f.write("}\n")  
 print(f"[+] DOT file generated: {output\_file}")  
  
def collect\_python\_files(directory):  
 python\_files = []  
 for root, \_, files in os.walk(directory):  
 for file in files:  
 if file.endswith(".py"):  
 python\_files.append(os.path.join(root, file))  
 return python\_files  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 code\_dir = "inventory\_codebase" # Update with your folder name  
 all\_calls = []  
  
 for file\_path in collect\_python\_files(code\_dir):  
 print(f"Analyzing {file\_path}")  
 calls = extract\_calls\_from\_file(file\_path)  
 all\_calls.extend(calls)  
  
 generate\_dot(all\_calls)

## 4.4 Sample Output (DOT Format)

The script generates a `.dot` file that can be visualized using Graphviz. Below is a sample of the DOT output generated by the tool:

digraph CallGraph {  
 main -> add\_item;  
 main -> update\_stock;  
 update\_stock -> validate\_item;  
}

## 5. Evaluation

### 5.1 Program Understanding Support:

* Call graphs helped track execution flow and redundant logic.
* Class/module diagrams highlighted tight coupling.
* Significantly enhanced maintainers' ability to navigate and comprehend code.

### 5.2 Maintenance Impact Evaluation:

We simulated a bug-fix in the stock update function:

* Without visualization: 2 hours to locate the bug.
* With visualization tool: 40 minutes (60% time saved).

### 5.3 Feedback Summary:

From a peer testing group of four:  
  
Evaluation Metric | Average Score (out of 5)  
--------------------------------|---------------------------  
Ease of Use | 4.5  
Visualization Clarity | 4.7  
Maintenance Relevance | 4.6  
Overall Usefulness | 4.8

## 6.Conclusion

The combination of RE and visualization is very useful for understanding the software and making it easier to maintain, especially in the case of legacy systems. Based on Graphviz, our approach yielded informative structural and behavioral diagrams which promoted faster debugging and better informed change decisions. This work also adheres to the principles of Lehman’s Laws and it may be considered a significant contribution to the practice of software maintenance.

## Contribution Table:

|  |  |
| --- | --- |
| Name & ID | Contribution |
| 24-93566-3 Lima , Rashida Begum | Introduction, Literature Review |
| 25-93648-1 Jahan, Nousheen | Literature Review,Framework Development |
| 24-93245-1 Chowdhury, Shah Fahim | Evaluation, Implementation |

#### **References and Citations**

[1] R. Koschke, "Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey," Journal of Software Maintenance and Evolution: Research and Practice, vol. 15, no. 2, pp. 87–109, 2003.

[2] B. Bellay and H. Gall, "A comparison of four reverse engineering tools," in Proc. 4th Working Conf. Reverse Engineering, 1997, pp. 2–11, doi: 10.1109/WCRE.1997.624571.

[3] C. Stringfellow, C. D. Amory, D. Potnuri, A. Andrews, and M. Georg, "Comparison of software architecture reverse engineering methods," Information and Software Technology, vol. 48, no. 7, pp. 484–497, 2006.

[4] G. C. Gannod and B. H. C. Cheng, "A framework for classifying and comparing software reverse engineering and design recovery techniques," in Proc. 6th Working Conf. Reverse Engineering (Cat. No.PR00303), 1999, pp. 77–88, doi: 10.1109/WCRE.1999.806949.

[5] I. A. Salihu and R. Ibrahim, "Comparative analysis of GUI reverse engineering techniques," in Adv. Comput. Commun. Eng. Technol.: Proc. ICOCOE 2015, Springer, 2016, pp. 295–305.

[6] P. Tonella, M. Torchiano, B. Du Bois, and T. Syst{"a}, "Empirical studies in reverse engineering: state of the art and future trends," Empirical Software Engineering, vol. 12, pp. 551–571, 2007.

[7] C. M. Cuillou, "The reverse engineering of computer software in Europe and the United States: A comparative approach," Colum.-VLA J. L. & Arts, vol. 22, pp. 533, 1997.

[8] F. Buonamici, M. Carfagni, R. Furferi, L. Governi, A. Lapini, and Y. Volpe, "Reverse engineering modeling methods and tools: a survey," Computer-Aided Design and Applications, vol. 15, no. 3, pp. 443–464, 2018.

[9] M. A. Zeeshan and S. S. Waris, "Reverse engineering application instruments and code reliability: A comparative study of tools," in Advances in Manufacturing Technology XXXIV, IOS Press, 2021, pp. 53–63.

[10] J. Koskinen and T. Lehmonen, "Analysis of ten reverse engineering tools," in Advanced Techniques in Computing Sciences and Software Engineering, Springer, 2009, pp. 389–394.

[11] J. Mattei, M. McLaughlin, S. Katcher, and D. Votipka, “A qualitative evaluation of reverse engineering tool usability,” in Proc. Annual Computer Security Applications Conf., Austin, TX, USA, Dec. 2022, pp. 1–13.

[12] D. Votipka, S. M. Rabin, K. Micinski, J. S. Foster, and M. L. Mazurek, “An observational investigation of reverse engineers’ processes,” arXiv, abs/1912.00317, Dec. 2019.

[13] S.-Y. Yu, Y. G. Achamyeleh, C. Wang, A. Kocheturov, P. Eisen, and M. A. Al Faruque, “CFG2VEC: Hierarchical graph neural network for cross‑architectural software reverse engineering,” arXiv, abs/2301.02723, Jan. 2023.