Islamic University Of Technology

Computer Science And Engineering

Programme: Software Engineering

Investigating the Applicability of Lehman's Laws of Software Evolution using Metrics: An Empirical Study on Open Source Software

---

SWE 4801
Assignment 01

Topic: Continuing Growth And Declining Quality

GROUP 12

Raiyan Noor, 190042116

Azmayen Fayek Sabil, 190042122

Abir Hossain 190042139

Contents

1 Introduction

In essence, software systems are inherently subject to changes to meet evolving functional and non-functional needs. The applicability of Lehman's Law of Software Evolution is assessed by studying the evolution of software systems in line with Lehman's principles. Lehman's Law suggests that software systems continually change and may witness a decline in quality over time. Researchers use empirical analysis to measure the degree to which software systems conform to Lehman's observations, taking into account factors like code complexity, maintenance efforts, and user satisfaction. Understanding the relevance of Lehman's Law in today's software development practices allows stakeholders to formulate strategies to counteract quality degradation and ensure the long-term sustainability of software systems.

Our study aimed to explore the applicability of Lehman's Law based on software quality metrics. It specifically focuses on two laws proposed by Lehman: **"continuing growth"** and **"declining quality."** These laws depict the propensity of software systems to become increasingly complex and to perpetually expand in size and functionality. By analyzing software quality metrics, we sought to determine whether these laws are valid in the context of contemporary software development practices.

2 Literature Review

Investigating the Applicability of Lehman's Laws of Software Evolution using Metrics: An Empirical Study on Open Source Software

This work is based on the paper "Investigating the Applicability of Lehman's Laws of Software Evolution using Metrics: An Empirical Study on Open Source Software". Lehman's Laws of Software Evolution describe how software systems evolve over time, suggesting that they must continually adapt to their environment, increase in complexity, grow continuously, and may experience a decline in quality.

The empirical study by Drouin and Badri explored the applicability of Lehman's laws using a synthetic metric called the Quality Assurance Indicator (Qi), which combines various object-oriented software attributes. They focused on two open-source Java software systems, collecting data for over four years for the first system and over seven years for the second. The aim was to see if the Qi metric could support the applicability of Lehman's laws, particularly in relation to continuous change, increasing complexity, continuous growth, and declining quality. The empirical results provide evidence that Lehman's laws were supported by the collected data and the Qi metric.

Another study by de Oliveira, de Almeida, and da Silva Gomes evaluated Lehman's laws within the context of Software Product Lines (SPLs). They used two techniques - the KPSS Test and linear regression analysis - to assess the relationship between Lehman's Laws and SPL assets. The results showed that three laws were supported based on the data used (continuous change, increasing complexity, and declining quality), while the other law (continuous growth) was partly supported, depending on the SPL asset evaluated.

In a paper by Johari, the authors conducted a study on two open-source software systems and applied Lehman's laws of software evolution using several metrics. They found that some laws, such as law 1, law 2, and law 6, were easily determined using the metrics. However, the applicability of law 3, law 4, and law 5 to open-source software systems was difficult to determine and would require more empirical studies with relevant data.

In conclusion, the literature suggests that Lehman's Laws of Software Evolution can be supported by empirical studies using metrics, especially when applied to open-source software systems. The synthetic metric Qi, for example, has shown potential in guiding quality assurance actions throughout the evolution of software systems. However, the findings are often specific to the software systems and metrics used in the studies, and more research is needed to draw broader conclusions.

3 Lehman's Law

Lehman's Laws of Software Evolution [1] are a set of principles that describe how software systems evolve over time. These laws were formulated by Lehman and Belady starting in 1974. Here are the eight laws they proposed:

1. **Continuing Change**: An E-type system must be continually adapted or it becomes progressively less satisfactory.
2. **Increasing Complexity**: As an E-type system evolves, its complexity increases unless work is done to maintain or reduce it.
3. **Self Regulation**: E-type system evolution processes are self-regulating with the distribution of product and process measures close to normal.
4. **Conservation of Organisational Stability (Invariant Work Rate)**: The average effective global activity rate in an evolving E-type system is invariant over the product's lifetime.
5. **Conservation of Familiarity**: As an E-type system evolves, all associated with it, developers, sales personnel and users, for example, must maintain mastery of its content and behaviour to achieve satisfactory evolution. Excessive growth diminishes that mastery. Hence the average incremental growth remains invariant as the system evolves.
6. **Continuing Growth**: The functional content of an E-type system must be continually increased to maintain user satisfaction over its lifetime.
7. **Declining Quality**: The quality of an E-type system will appear to be declining unless it is rigorously maintained and adapted to operational environment changes.
8. **Feedback System**: E-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base.

These laws provide enduring insights into the nature of software systems and their evolution. They are particularly relevant in today's software development landscape, where agility and adaptability are paramount.

4 Methodology

In conducting this analysis, we followed a comprehensive methodology to assess the React library repository's evolution and adherence to Lehman's law of continuing growth and declining quality. The following steps elucidate the technical details of our approach:

4.1 Repository Selection

- **Choice of React:** Selected the React library repository for in-depth analysis.
- **Motivation:** Motivated by its widespread usage and active maintenance within the open-source community.
- **Significance:** The project's significance and diverse contributions played a crucial role in the selection, ensuring a representative case for our study.

4.2 Data Collection

- **SonarQube Usage:** Employed SonarQube for static code analysis, leveraging its capabilities for assessing code quality.
- **Systematic Exploration:** Systematically explored each branch, version, and release of the React repository to capture a comprehensive view of its evolution.
- **Assessment Metrics:** Assessed various software metrics, including code complexity, duplication, code smells, and other quality-related indicators, providing a holistic understanding of the project's codebase health.

4.3 Compilation of Metrics

- **Structured Data Repository:** Collated acquired software metrics into a structured JSON-formatted file.
- **Comprehensive Data Repository:** This file served as a comprehensive data repository encapsulating the multifaceted characteristics of the React project across its various iterations, forming the basis for our subsequent analyses.

4.4 Data Extraction and Analysis

- **Python Utilization:** Leveraged the Python programming language for detailed processing of the JSON-formatted file, ensuring efficient extraction and manipulation of relevant metrics.
- **Focus on Lehman's Law:** Focused on specific metrics relevant to Lehman's law, such as the growth of codebase size, changes in code complexity, and the emergence of code quality issues, allowing us to draw meaningful insights from the data.

4.5 Graphical Representation

- **Library Utilization:** Utilized Python libraries, including Matplotlib, to transform the extracted metrics into visually appealing graphs.
- **Facilitated Exploration:** Graphical representations facilitated a nuanced exploration of the React repository's evolutionary patterns and their implications on software quality, making complex data more accessible for interpretation.

This methodical approach not only facilitated a granular examination of the React repository's evolution but also provided a technical foundation for understanding the interplay between growth dynamics and code quality degradation as posited by Lehman's law.

5 Metrics Used for Analysis

5.1 Metrics that suggest growth:

- **number Of files**: This metric represents the total number of files in a software project. It can give an idea about the size and complexity of the project.
- **number of classes**: This metric counts the number of classes in a software project. In object-oriented programming, a class is a blueprint for creating objects. A high number of classes might indicate a high degree of modularity, but it could also suggest unnecessary complexity.
- **number of functions**: This metric counts the number of functions or methods in a software project. A function is a group of related statements that perform a specific task. A high number of functions might indicate a high degree of modularity, but it could also suggest unnecessary complexity.
- **NCLOC**: This stands for "Non-Comment Lines of Code". It measures the number of lines that are neither blank nor comments. This metric is often used to estimate the amount of effort that will be required to develop or maintain the software.

5.2 Metrics that suggest quality:

- **major violations**: This metric represents the number of major issues or breaches in the code that violate the coding standards or best practices. Major violations often have a significant impact on the quality of the code and may lead to serious problems such as bugs, security vulnerabilities, or maintainability issues.
- **code smells**: Code smells are indicators of potential problems in the code. They are not bugs, but they suggest weaknesses in the design that may slow down development or increase the risk of bugs or failures in the future. Examples of code smells include long methods, large classes, duplicate code, and so on. The "smelliness" or code smell density can be calculated by dividing the number of code smells by the total number of lines of code in an application or module.
- **critical violations**: Critical violations are severe coding errors that need immediate attention. They often represent serious issues in the code that can lead to critical system failures or major security vulnerabilities. These are the issues that should be addressed as a top priority due to their potential impact on the system.

6 Results and Findings

6.1 Individual Results

Individual Metric Assessment: Below are the visual representation of each metric we analysed over each release.

1. **Files:** Number of files always follows an increasing growth pattern. Thought it is not in a predictable fashion but the files keep increasing in a codebase over time.

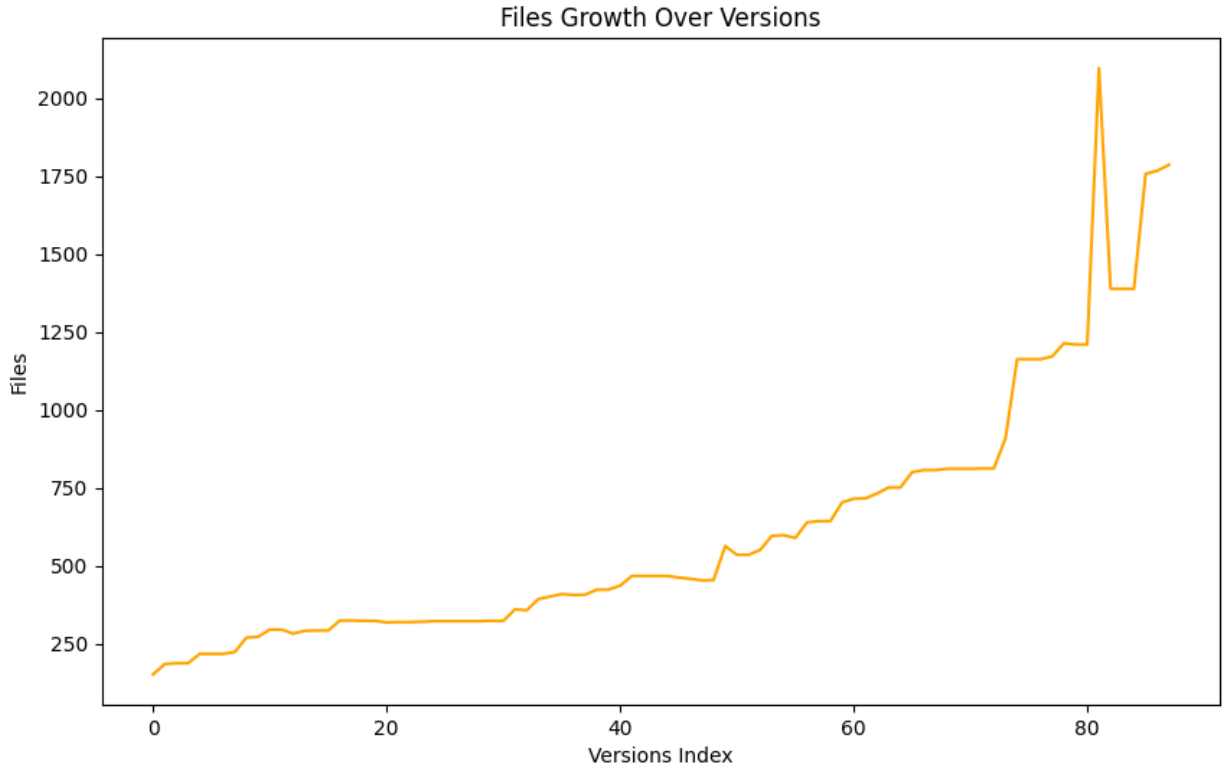

Figure 1: **Files Growth Over Time**

2. **Functions:** Functions follow an increasing trend but more than the previous metrics. Functions tend to grow more rapidly compared to classes and sometimes may be prone to decrease.

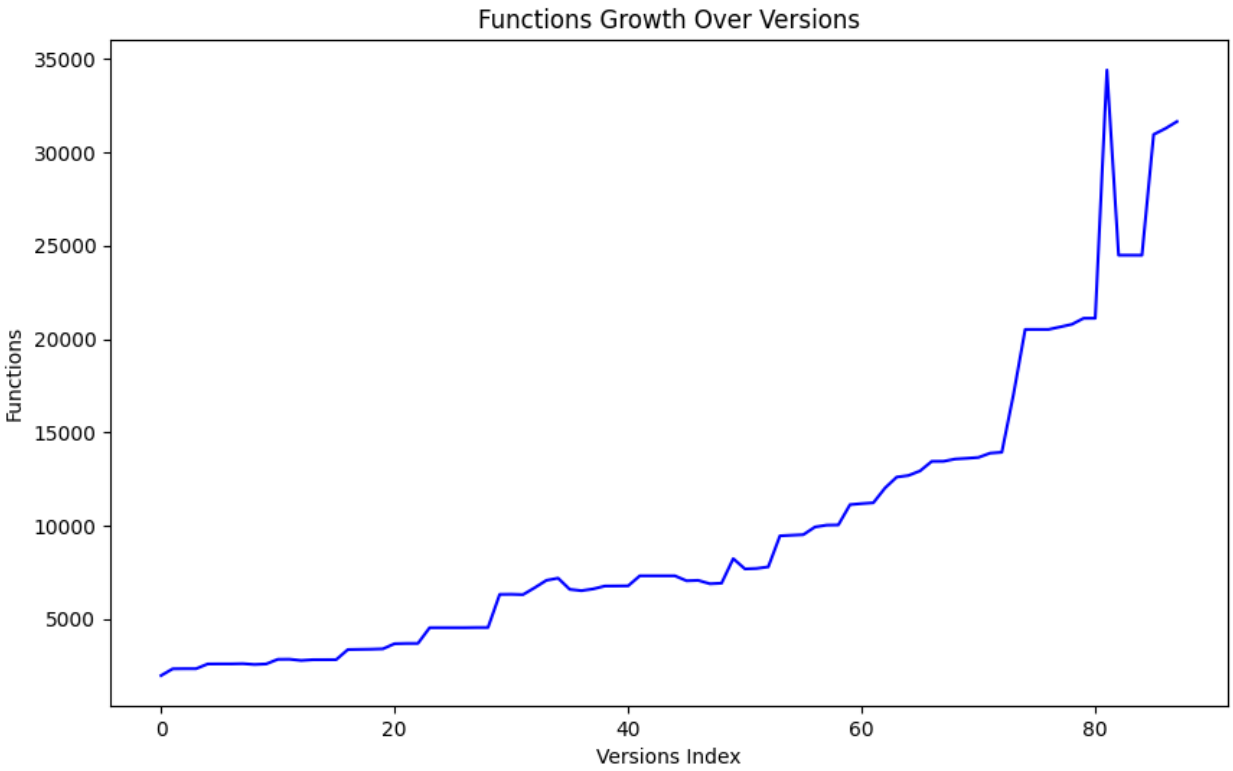

Figure 2: **Functions Growth Over Time**

3. **Ncloc:** Ncloc(Non-commenting Lines of Code) follows the most rapid growth of all the metrics. It is very rare for it t ever decrease. It mostly follows a rapid growth trend with each release.

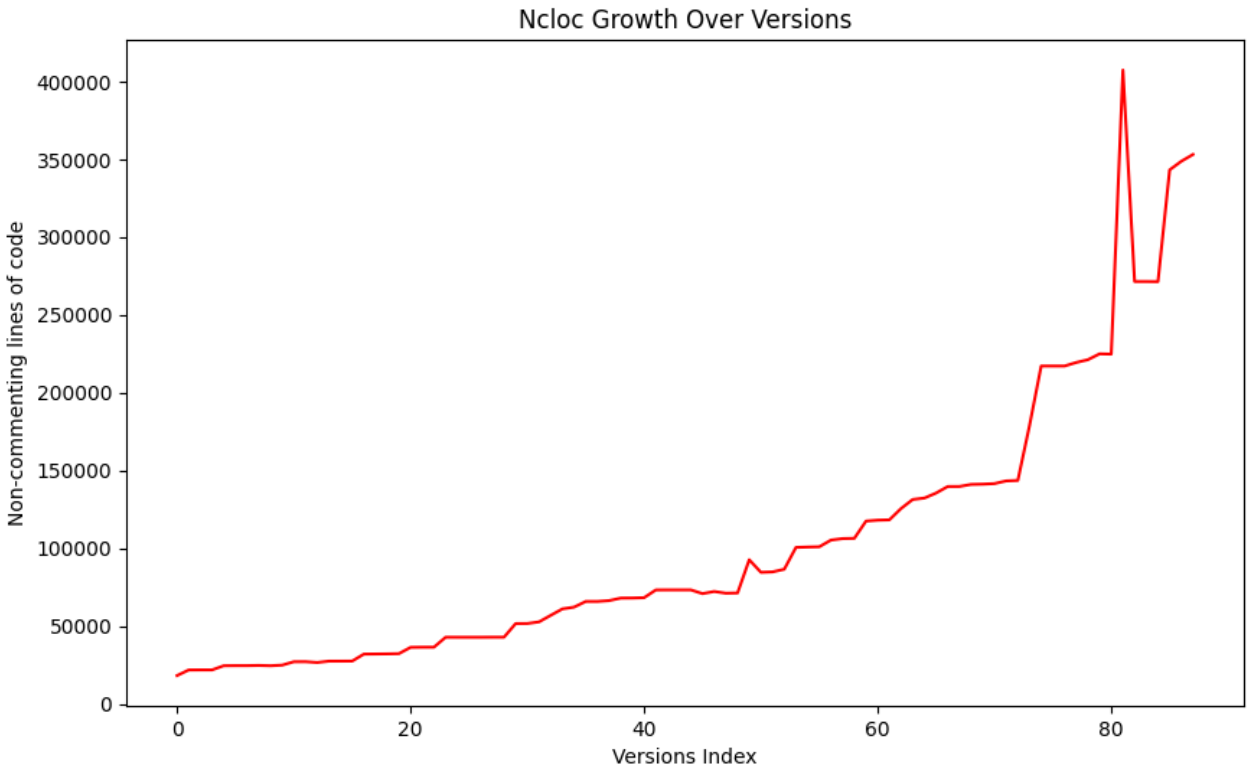

Figure 3: **Ncloc Growth Over Time**

4. **Major Violations:** Major Violations refer to the projects declining quality. With each release to a codebase the number of major violations is increasing proving the codebase's quality is decreasing. It also follows the same trend here.

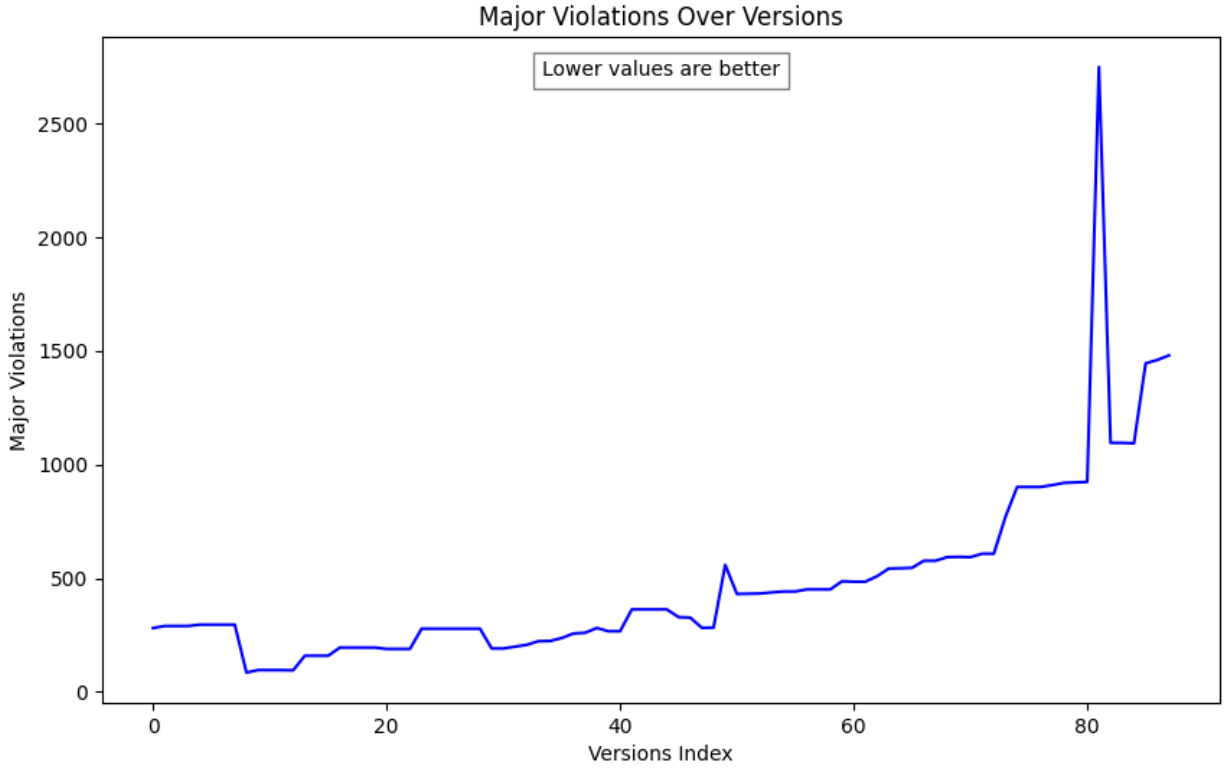

Figure 4: **Major Violations Over Time**

5. **Code Smells:** Code smells are an easy indicator for code quality. Increasing code quality usually means declining quality. Here the code smells increase for most releases. But then For some release it suddenly drops and then it continues the upward trend again.

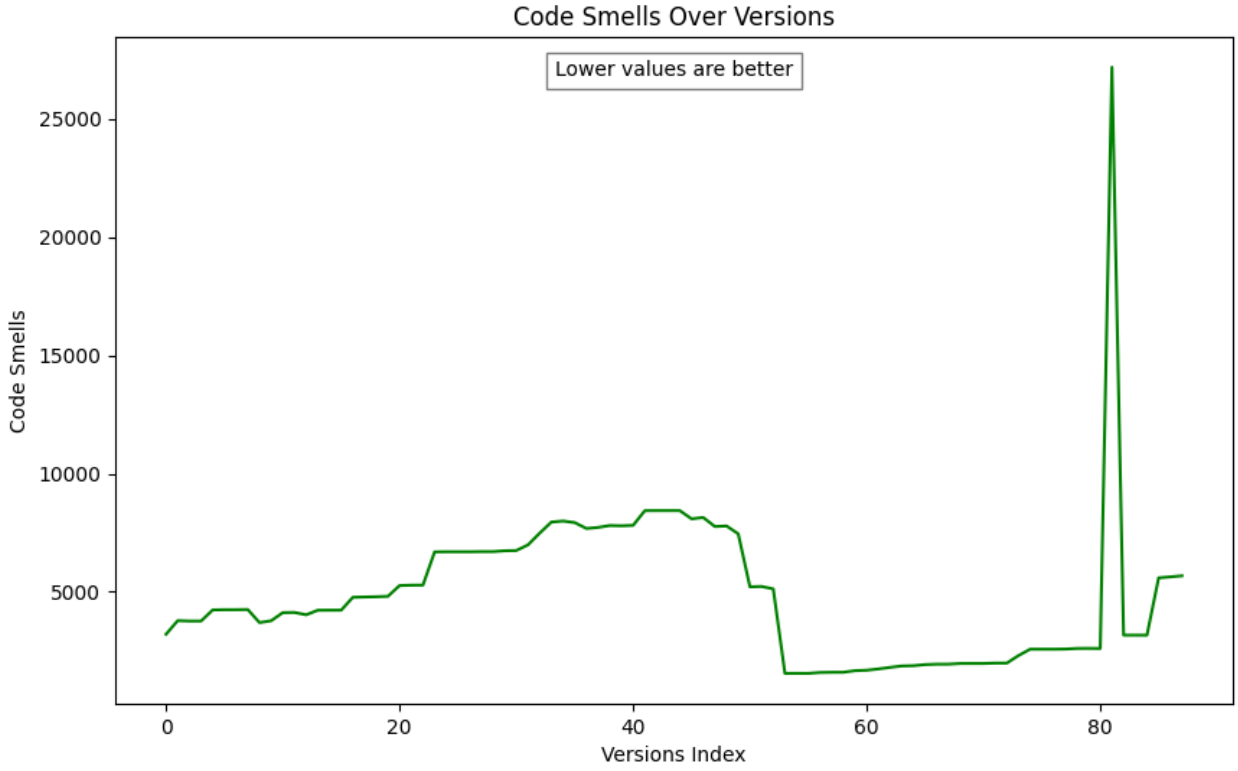

Figure 5: **Code Smells Over Time**

6. **Critical Violations:** Critical violations also follow the same trend of growth as previous metrics. For most releases it follows an upward growth indicating declining quality but for a certain release it suddenly decreases followed by another upward trend and a sudden boom which later normalizes to a upward trend again.

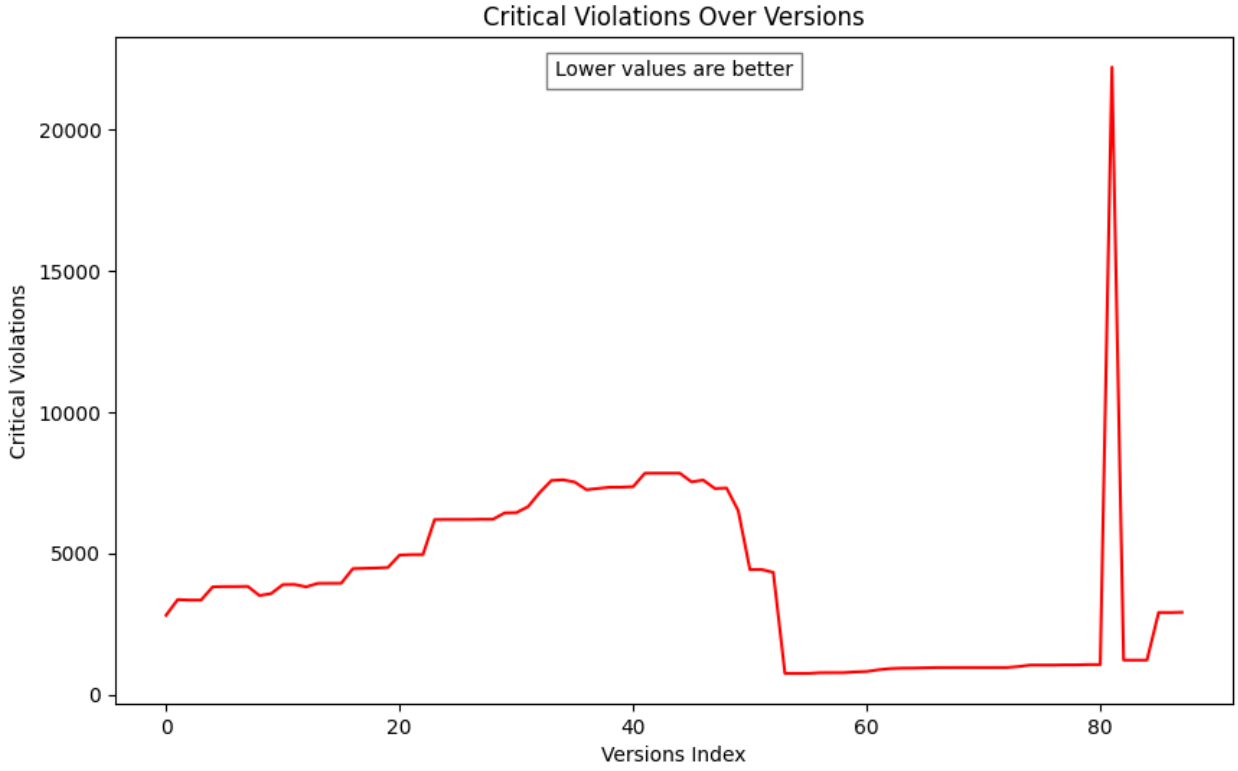

Figure 6: **Critical Violations Growth Over Time**

6.2 Findings

We found that the codebase **does indeed follow Lehman's law of continuing growth and declining quality**

- **Growth Trends:** Out of the 4 metrics we considered to find growth trends, we found that all the metrics follow some growth trend. We have found that over time the codebase continues to grow, this is consistent in almost all the growth metrics we have tested. The number of functions also grew very slowly. But the number of NCLOC(Non-commenting Lines of Code) grew rapidly.

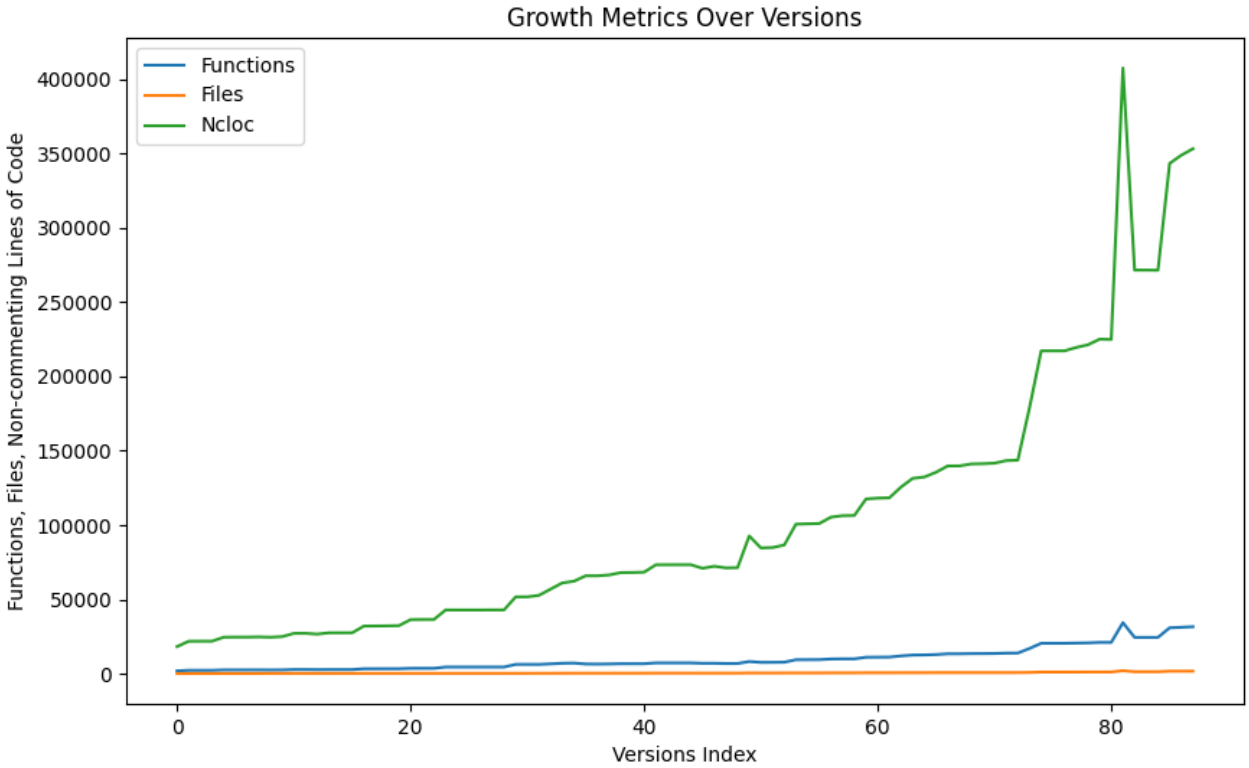

Figure 7: **Growth Over Time**

- **Quality Assessment:** We have also found that code quality tends to decrease as time progress. We have found that for most releases code quality tends to decrease but in some releases they rapidly increased code quality but then some releases very rapidly decreased the code quality.

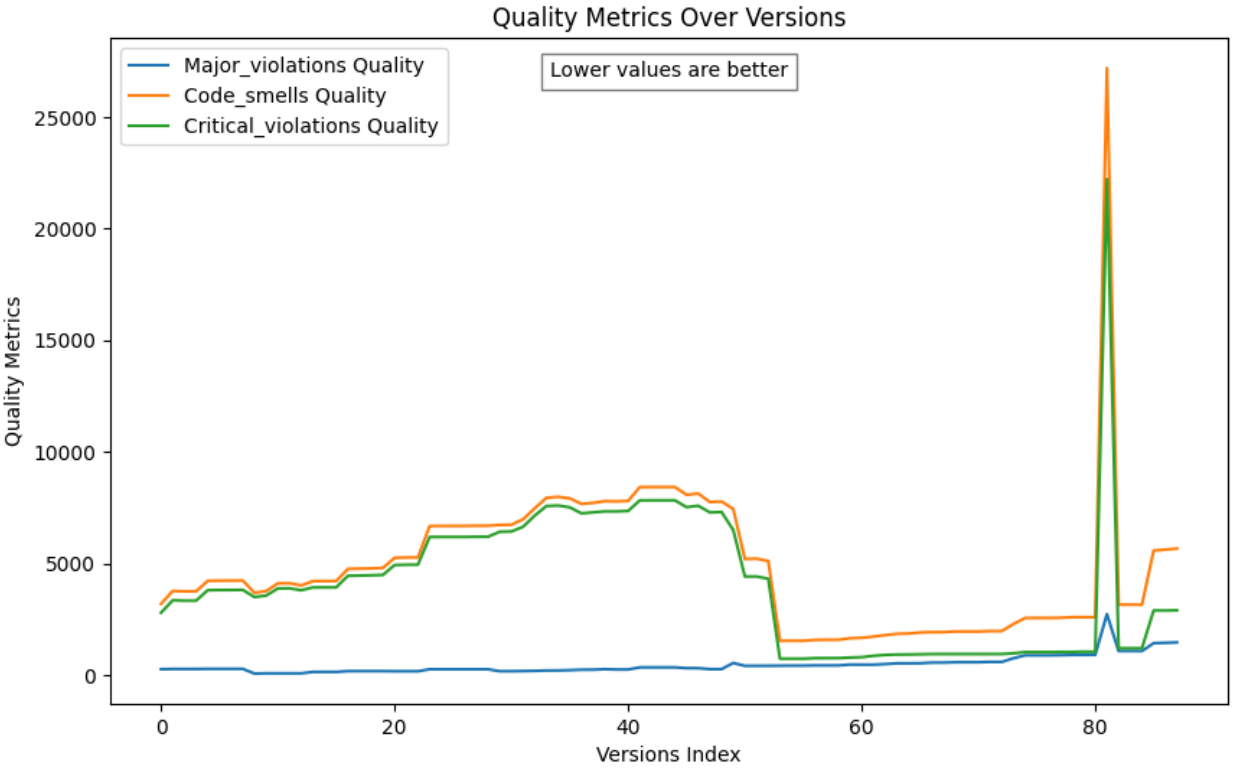

Figure 8: **Quality Over Time**

- **Interplay Between Growth and Quality:** We can see while growth always follow an increasing trend, that is not true for quality. Code quality tends to decrease in most releases as complexity increases. But now and then there will be releases where code quality might decrease significantly and the opposite also holds true. There might be releases where code quality suddenly increase to a significant amount.

7 Conclusion

The analysis of the React library repository's evolution has provided valuable insights into its adherence to Lehman's law of continuing growth and declining quality. The graphical representations of key metrics align with the expected patterns proposed by Lehman's law.

Continuing Growth: The observed trends in codebase size, measured by metrics like lines of code and file count, indicate a continuous expansion of the React project over its various iterations. This growth may be attributed to the project's increasing functionality, feature additions, and ongoing development efforts.

Declining Quality: Metrics related to code quality, such as code smells, major violations, and critical violations, demonstrate a discernible decline over time, supporting the notion of diminishing software quality as the project evolves. This decline may be indicative of accumulating technical debt and challenges associated with maintaining high-quality code in a rapidly evolving codebase.

Observations:

- *Code Smells and Violations:* The increasing count of code smells and violations suggests a potential need for refactoring and code review to address emerging issues and maintain code health.
- *Proactive Quality Measures:* The findings underscore the importance of proactive measures, such as regular code reviews, automated testing, and adherence to coding standards, to mitigate the impact of declining quality over time.
- *Balancing Growth and Quality:* Striking a balance between the continuing growth of the codebase and maintaining high software quality poses a significant challenge. It necessitates strategic decision-making and continuous efforts to enhance development processes.

The convergence of these findings substantiates the applicability of Lehman's law to the React library repository. As the project continues to grow, there is an observable trade-off with declining code quality. This analysis underscores the importance of ongoing vigilance and proactive measures to maintain or enhance software quality in large and continuously evolving codebases.

Bibliography

[1] N. Drouin and M. Badri, "Investigating the applicability of the laws of software evolution: A metrics based study," in *Evaluation of Novel Approaches to Software Engineering*, J. Filipe and L. A. Maciaszek, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 174–189.