Violations of SOLID Principles in Real-World Libraries: A Case Study on Java Collections

Shriram Kalyan Patil
Master of Computer Application,
Sinhgad Institute of Business Administration and Research,
Pune - 411048
patilshriram200@gmail.com

Umakant Shriram Kesare
Master of Computer Application,
Sinhgad Institute of Business Administration and Research,
Pune - 411048
kesareuma@gmail.com

Dr. Sharda Patil, Ph.D.
Associate Professor,
Master of Computer Application,
Sinhgad Institute of Business Administration and Research,
Pune - 411048
santosh.sharada@gmail.com

Abstract

SOLID principles are at the core of object-oriented software design, supporting systems to work in modularity, extendibility & maintainability. Nevertheless, most advanced and common frameworks, like the Java Collections Framework (JCF), periodically violate these principles, due to legacy design choices, backward compatibility issues, and platform demands. This study examines such deviations from the JCF model, and finds and analyzes specific cases (e.g. usage of inheritance in Stack and Properties classes and behavioral inconsistency in unmodifiable collections). Here, we have evaluated how these violations impact software maintainability, extensibility, and developer understanding. It also explores the design and engineering tradeoffs that contributed to these deviations and their lasting consequences for sustainable software engineering. In this paper, we propose refactoring strategies and design recommendations to improve library design through SOLID principles.

1. Introduction

The principles of design in software engineering today are to achieve flexibility, maintainability, and re-usability of the software. SOLID stands for Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion among others are the basis of a good object-oriented design [5]. They offer developers a set of guidelines for creating modular, testable and easily extendable systems.

· Single Responsibility Principle (SRP): Each class should have only one reason to change**[8]**.

· Open/Closed Principle (OCP): Software entities should be open for extension but closed for modification**[8]**.

· Liskov Substitution Principle (LSP): Sub-classes must be substitutable for their base classes without altering program correctness**[8]**.

· Interface Segregation Principle (ISP): Clients should not be forced to depend on methods that they do not use**[8]**.

· Dependency Inversion Principle (DIP): High-level modules should depend on abstractions rather than concrete implementations**[8]**.

The SOLID principles are a good idea, but their violations seem to be rampant in still a lot of well-known libraries such as Java Collections Framework (JCF) even after being considered as a best practice. Most of these violations are historical limitations, backwards compatibility and compromises made for usability. One example is that Stack class inherits from Vector, causing violation to the Liskov Substitution Principle (LSP) since a stack should not provide random-access operations for vector. Likewise the Properties class extends Hashtable, which would violate SRP and LSP by permitting mixed data types and enforcing configuration-specific behavior on a general-purpose collection. Some such immutable collection wrappers also like Collections.unmodifiableList(), break the behavior contracts, resulting in implicit design infractions

Being aware of these infractions is critical since the Java Collections Framework is widely used in professional software and educational settings. While legacy limitations sometimes serve as justification for these design flaws, they sometimes result in confusion among students, decreased maintainability and dissemination of bad design.

This study systematically analyzes violations of the SOLID principles in the Java Collections Framework, explores their underlying causes, assesses their implications for software design, and proposes recommendations for achieving stronger adherence to these foundational design principles in future API development.

1.1 Statement of the problem

The aim of the SOLID principles is to create software that is robust, maintainable, and extensible. As models representing optimal practices of Java, however, the Java Collections Framework contains several failings which violate these principles. These flaws, which stem from backward compatibility and historical evolution, detract from the clarity and maintainability of the software.

For example, when a stack behaves as a vector it clashes with its theoretical promise to be a restricted LIFO structure. Similarly, Properties extends the Hashtable which allows heterogeneous key-value pairs that violates both encapsulation and type safety. And this is illustrative of the fact that even the core libraries fail a great deal of the design values they were supposed to teach.

The problem is twofold:

Technical impact: Developers who use these APIs are forced to grapple with conflicting behaviors that add to their cognitive resources and the risk of misuse [7].

Educational impact: While these libraries are extensively taught at the higher educational institutes in Computer Science, misconceptions about good software design are propagated when the library design is in question [6].

Therefore it is necessary to systematically investigate these violations and extract insights that will give direction for the further development of future frameworks.

1.2 Objective of the Study

This study seeks to achieve the following objectives:

- To **understand** the significance of SOLID Principles in Object-Oriented design.
- **To uncover** actual SOLID principle violations as they can be found in the Java Collections Framework.
- **To explain** the impact of these violations on the code quality, readability and maintainability.
- **To investigate** the engineering and compatibility trade-offs that brought such violations upon themselves.
- **To propose** improved design approaches or refactoring strategies that are aligned with the SOLID principles.
- **To suggest better design** tactics or refactoring methods based upon SOLID principles. This project contains practical and educational applications for developers and library developers.

1.3 Hypothesis of the Study

H_1: The Java Collections Framework exhibits multiple violations of the SOLID principles, in the context of the SRP and LSP.

H_2: These violations negatively affect software maintainability, and long-term usability.

H_3: Backward compatibility and legacy design limitations are the main reasons for the continued existence of these breaches.

H_4: The presence of such violations in a commonly used framework affects the developers' interpretation and practical implementations of SOLID principles.

2. Review of Literature :

2.1 Overview of SOLID Principles

The SOLID principles are a collection of five design guidelines that aim to contribute towards making software systems more maintainable, flexible, and easy to understand. First popularized by Robert C. Martin (or "Uncle Bob") [21], these principles are extensions of the original ideas of object-oriented programming, promoting

modular behavior and low coupling among classes.

- Single Responsibility Principle (SRP): A class should have only one reason to change **[8]** . It fosters cohesion by guaranteeing that every class does one well-defined task.
- Open/Closed Principle (OCP): Software entities should be open for extension but closed for modification **[8]** . This enables new behaviors to be added via inheritance or composition without modifying the existing code.
- Liskov Substitution Principle (LSP): Objects of a subclass should be replaceable with objects of their superclass without affecting program correctness **[8]** .
- Interface Segregation Principle (ISP): Clients should not be forced to depend on methods that they do not use. This discourages the use of large multipurpose interfaces **[8]** .
- Dependency Inversion Principle (DIP): High-level modules should depend on abstractions rather than concrete implementations **[8]** .

These principles are not "hard" rules but design heuristics to aid developers in creating strong and maintainable structural designs. However, actual software frameworks sometimes fail to address some of these due to their compatibility with traditional systems such as older systems, backward compatibility or performance issues.

2.2 Violations of SOLID Principles in Java Collections Framework

- *Stack and Liskov's Substitution Principle (LSP) Violation*

A typical case of Liskov's violation is that of the class that extends Vector; Stack. Based on the LSP, a subclass (stack) needs to be usable wherever the parent class (vector) is needed in the program without affecting the correctness of the program. However, since Vector provides operations such as insertElementAt() or removeElementAt() that break the LIFO (Last In First Out) semantics of a stack, substitutability breaks [22].

```
Vector<Integer> v = new Stack<>();
v.add(0, 100); // Violates stack behavior by inserting at index 0
```

This misuse of inheritance creates behavior inconsistencies. Instead, a better design would have been composition (a stack with a vector) rather than inheritance.

- *Properties and Single Responsibility Principle (SRP) Violation*

The Properties class in Java simply extends the Hashtable<Object, Object> which violates both SRP and LSP.
Although on the other hand, the Hashtable is a very general-purpose key–value data structure, Properties focuses on tuning the configuration settings, e.g., string-based keys and values[23]. However, because of inheritance, developers are allowed to keep arbitrary objects in a Properties object, which can lead to type and semantic mistakes.

```
Properties props = new Properties();
props.put(10, true); // Compiles, but violates intended design
```

This violates SRP because the class is required to perform configuration management and general-purpose data storage, and also breaks LSP because a Properties object cannot replace a Hashtable without causing type-related problems.

- *Collections.unmodifiableList() and Behavioral Substitution*

The Collections.unmodifiableList() method gives a read-only view of a given list, which appears to be in line with OCP and LSP at first, but it leads to behavioral substitution problems if an implementation of the List interface is shown due to the returned value yielding UnsupportedOperationException when modifying [9].

```
List<Integer> list = Collections.unmodifiableList(Arrays.asList(1, 2, 3));
list.add(4); // Throws UnsupportedOperationException
```

This violates the Liskov Substitution Principle, where the client expects that any List must support the add() function. This design approach improves the immutability of the code, but it introduces confusion and

violates the substitutability from a more detailed design angle.

- *Enumeration and Open/Closed Principle (OCP) Concerns*

Older JCF classes like Vector and Hashtable depend on the Enumeration interface, which is older than the new Iterator pattern; the compatibility of the two interfaces (Enumeration and Iterator) demonstrates how backward compatibility can limit adherence to OCP. Instead of adding features that would extend or change existing code, the JCF preserved both interfaces and resulted in a degree of design redundancy and increased maintenance complexity.

2.3 Consequences of SOLID Violations in JCF

The violations discussed, although quite subtle, have enduring implications for software quality and developers' experience:

- Reduced Maintainability: Inheritance-based coupling makes changes expensive and difficult.
- Behavioral Inconsistencies: Developers encounter runtime errors due to substitutability violations in unexpected ways.
- Conceptual Confusion: Beginners usually simply imitate bad patterns as a matter of course for learning Java Collections.
- Type Safety Issues:Classes like Properties undermine Java's strong typing guiding principles.
- Increased Complexity: Backward compatibility creates class hierarchies and a mix of responsibilities
- Such outcomes emphasise the tradeoffs between clean design and functional usability in the evolution of libraries [7].

2.4 Gaps in Existing Literature

Despite the significance of SOLID principles in software engineering, small amounts of literature have focused on their violations in standard libraries like JCF.

Most prior research has focused on the following :

- Automated SOLID compliance detection.
- Case studies of object-oriented design patterns.
- Static code analysis for maintainability.

However, very limited research has been done on the following:

- In JCF examines specific SOLID violations.
- Historical and design context explain the cause of these problems.
- Provides practical refactoring approaches for better SOLID compliance

Filling this gap holds promise to enhance academic knowledge and enhance practical improvements in implementation of Java library design.

3. Research Methodology

3.1 Research Approach

This study is qualitative case study study with the objective to investigate in detail how the Java Collections Framework (JCF) breaches or adheres to the SOLID design principles, that is relevant, as rather than using numerical data, the study examines concepts, codes, and interpret, which allows detailed exploration of the classes and components of the JCF presenting real world design problems [1].

This report does not intend to critique the JCF, instead it intends to emphasize design trade-offs and legacy restrictions that affected non-SOLID compliance. Examining various examples, the study aims to draw insights gleaned from both library designers and software engineers.

3.2 Data Collection Method

This study utilized a qualitative analysis method based on direct observation of source code and documentation of the Java Collections Framework (JCF).

- Class hierarchies, method definitions, inheritance, and inheritance relationship descriptions from Java 8 and 11 API documentation.
- Source code from OpenJDK to determine concrete design and implementation decisions.
- To understand the proposed design rationale, we used official Java tutorials and Oracle documentation.
- Other secondary sources, like academic publications, model and design pattern

references and software engineering books, were cited for a conceptual background in SOLID principles.

Each class was examined to see if its design reflected or failed to comply with some SOLID principles. Further, they were categorized by the type of violation detected and the principle violated, with their likely cause of design violations (e.g., backward compatibility, historic design or abuse of abstraction).

3.3 Data Analysis Method

Comparative analysis was carried out with:

The conceptual description of the SOLID principles, and The code implementation behavior within Java Collections Framework.

The process was performed in the phases of Identification of Candidate ClassesClasses suspected of violating SOLID principles (e.g., Stack, Properties, Vector, Hashtable, Collections.unmodifiableList) were selected for further in-depth analysis.

Principle-based classification: We screened each class for violations of the principles for SOLID — such as SRP, OCP, LSP, ISP or DIP.

Code and Behavior Analysis Segments of source code, class relationship and interface behavior were examined to inspect for patterns (example, inheritance abuse, error of type for example, or non-conformity in behavioral contract).

Impact Analysis Categorical consequences of each violation have been evaluated related to maintainability, reusability, readability.

Refactoring Proposals Based on each observed violation recommendation of refactoring tactics (e.g., composition, interfacing, abstraction) is suggested to improve its compliance on SOLID.

4. Proposed Refactoring Solutions

Several recommended refactoring ways for combating the observed violations have been proposed. These are designed to make the JCF classes fit into SOLID principles. In essence, they are made in the hope that they will be able to help bring them up to code correctly while maintaining their backward compatibility where possible.

5.1 Refactoring the Stack Class:

Problem: Inherits from Vector, hence violates LSP and SRP.
1. First solution : Composition and not inheritance. The Stack class will use a List or Deque internally for writing to storage instead of extending Vector class.

```
public class Stack<T> {
    private final List<T> elements = new ArrayList<>();

    public void push(T item) {
        elements.add(item);
    }

    public T pop() {
        if (elements.isEmpty()) throw new EmptyStackException();
        return elements.remove(elements.size() - 1);
    }
}
```

This allows the Stack to have one responsibility (LIFO control), and we don't introduce non-essential vector methods (like insertElementAt()).

2. Second solution : Use the Deque interface (e.g., ArrayDeque) to demonstrate stack behavior; that uses composition and its built-in LIFO methods (push(), pop(), peek())--simple and type secured.

```
   /*
Java itself recommends this!
Stack (legacy, since JDK 1.0) is discouraged.
Instead, use Deque (ArrayDeque or LinkedList)
which naturally supports LIFO:
*/

    public void dequeueStack(){
        Deque<Integer> stack = new ArrayDeque<>();
        stack.push(10);
        stack.push(20);
        System.out.println(stack.pop()); // 20
    }
```

5.2 Refactoring the Properties Class:

Problem: Extends Hashtable<Object, Object>, which violates SRP and LSP.
Solution: Require String key-value pairs in composition and through type-safe generics.

```
public class Properties {
    private final Map<String, String> config = new HashMap<>();

    public void setProperty(String key, String value) {
        config.put(key, value);
    }

    public String getProperty(String key) {
        return config.get(key);
    }
}
```

This design keeps responsibility (configuration management) as clear as possible and removes mixing of the unsafe types thereby making the maintainability and readability better.

5.3 Redesigning Unmodifiable Collections

Problem: The Collections.unmodifiableList() method breaks LSP by returning a list implementation that does not fully support the List interface.
Solution: Add ReadOnlyList interface which defines read-only operations.

```
public interface ReadOnlyList<T> {
    T get(int index);
    int size();
    boolean contains(Object o);
}
```

This isolates mutable behaviors and immutable behaviour in turn, keeping contracts more or less in lockstep so that contracts are decoupled and predictable whilst maintaining the OCP (Open for extension, closed for modification).

5.4 General Recommendations

- Prefer composition instead of inheritance in order to improve cohesion and reduce coupling.
- Explicit abstractions for immutable structures are added to avoid behavioral mismatches.
- Encourage backward-compatible refactoring using adapter patterns , where direct redesign is impractical.
- Enable backward-compatible refactoring with the help of adapter patterns, when redesign is impractical.Improve documentation to identify design exceptions and deliberate infractions of the SOLID principles.
- Use automated static analysis to find in large codebases SOLID violations.

5. Limitations of the study

This research indicates a clear path that leads the Java Collections Framework (JCF) away from the SOLID design principles but at the same time, there are certain limitations that need to be established so that the findings could be interpreted accurately:

- Limited Scope of Analysis:
  The study only considers three classes of the JCF (namely Stack, Properties, and immutable collection wrappers). The other parts of the Java API or the independent libraries outside the Java API were not reviewed and could also have different design flaws, or both are not considered.
- Qualitative Nature of the Study:
  The analysis is qualitative in nature and mainly conceptual; it is analysis that consists of code inspection and design study rather than empirical measurement like defect rate, performance benchmarks, or user studies. Accordingly, the results will emphasize design reasoning, not quantitative assessment.
- Version Dependency:
  Data from OpenJDK (JDK 21) are used for this analysis. The next versions of Java might refactor or modify these classes, which could make the relevance of particular violations significantly change.
- Subjectivity in Interpretation:
  Determining whether (or when) a class violates a SOLID principle is subjective. Depending on the understanding from research on design intent and practical trade-offs, different researchers can

potentially have slightly different views to do with the same code.

- Lack of Tool-Based Validation: Although SonarQube was among the static analysis tools, this paper does not introduce automated detection metrics nor, through this methodology, significant validation at the large scale of many projects.

However, in spite of these limitations, the analysis identifies ubiquitous design trade-offs, which continue to impact the fundamental architecture of Java on the whole and lays a ground for further empirical study.

6. Future Scope

Our current study provides many directions for future study and development. These recommendations may contribute to reinforce awareness and enhance the compliance of SOLID with software frameworks:

- Automated SOLID Compliance Detection: There is hope by working on automated solutions to detect SOLID compliance violations in large Java codebases. Tools of this kind could leverage static analysis, AST parsing, or machine learning techniques to uncover design anomalies.

- Quantitative Evaluation: For researchers to further enhance this work, it would be beneficial to gather empirical metrics (maintainability index, coupling, cohesion, etc.) as they quantitatively measure the effect of SOLID compliance or violation on code quality.

- Broader Framework Comparison: Comparison across some standard libraries such as .NET Collections, C++ STL, or Python's Collections module can help you see if there are similar design trade-offs between other ecosystems.

- Developer Awareness and Education: Future studies may survey both developers and students to examine whether they are impacted by being exposed to an imperfect version of the library's design while learning object-oriented principles.

- Proposing Refactored API Models: Building on what we've learned, scholars can code SOLID-compliant prototypes of existing JCF classes with the aid of composition, generics, or design patterns to prove improved maintainability.

7. Conclusions

Our results validate that the Java Collections Framework (JCF), despite developing to a certain extent and getting popular, contains a number of shortcomings of the SOLID principles, in particular the Single Responsibility Principle (SRP) and the Liskov Substitution Principle (LSP),so it's supporting Hypothesis H_1. We found that these violations occurred in the Stack, Properties, and Collections.unmodifiableList() classes, in which design integrity is impaired by improper inheritance, type inconsistency, and behavioral substitution issues

Thus consistent with Hypothesis H_2, these violations have a negative impact on the maintainability, readability, and long-term use of the software. For example, misusing inheritance in Stack and Properties generates coupling and misunderstanding, and lack of a coherent behavior in unmodifiable collections leads to runtime errors and fuzzy conceptual understanding among developers.

In addition, the results support H_3 by demonstrating that backward compatibility and legacy design are the most common causes of these violations. Because of the historical development of the JCF and the requirement to maintain compatibility with the previous versions, structural reorganization was not possible, and functional and design issues were compromise-based.

Thirdly, reinforcing the hypothesis H_4, the research demonstrates the educational significance of such design weaknesses as the JCF also serves to teach concepts associated with object oriented programming. Misconceptions about good design techniques can arise from such infringements if they aren't adequately considered. Educators and curriculum developers need to remind students to reference these exceptions for the purpose of explaining SOLID as a strategy, in order to avoid misconception among students.

On the whole, the study emphasizes the need for regular re-evaluation of existing libraries like JCF based on changing standards in software design. Promoting that a composition rather than an inheritance of code elements would be adopted, the use of type-safe abstractions, and the encouragement of responsible developer behaviour will enable the next Framework to incorporate more of the SOLID approach, improving the quality of codes and instruction in a sound design.

8. References

- ***Journal of Systems and Software*** Volume 220, February 2025, 112254 .
- Abid, C., Alizadeh, V., Kessentini, M., do Nascimento Ferreira, T., Dig, D., 2020. 30 years of software refactoring research: A systematic literature review. arXiv: 2007.02194.
- Ampatzoglou, A., Chatzigeorgiou, A., 2007. Evaluation of object-oriented design patterns in game development. Information and Software Technology 49, 445–454. https://doi.org/10.1016/j.infsof.2006.07.003
- Erich Gamma,Richard Helm ,Ralph Johnson,John Vlissides, 1994. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional.
- Martin, R.C., 2000. Design Principles and Design Patterns 34.
- Pillay, N., 2010. Teaching Design Patterns
- Ramasamy, S., Jekese, G., Hwata, C., 2015. Impact of Object-Oriented Design Patterns on Software Development. International Journal of Scientific and Engineering Research Volume 3, 6.
- https://www.researchgate.net/publication/273451390_SOLID_Principles_in_Software_Architecture_and_Introduction_to_RESM_Concept_in_OOP .
- Abid, C., Alizadeh, V., Kessentini, M., do Nascimento Ferreira, T., Dig, D., 2020. 30 years of software refactoring research: A systematic literature review. arXiv: 2007.02194.
- AlOmar, E.A., AlRubaye, H., Mkaouer, M.W., Ouni, A., Kessentini, 2021. Refactoring practices in the context of modern code review: An industrial case study at X erox. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice. ICS
- AlOmar, E.A., Mkaouer, M.W., Ouni, A., 2024. Behind the intent of extract method refactoring: A systematic literature review. IEEE Trans. Softw. Eng. 50 (4), 668–694. http://dx.doi.org/10.1109/TSE.2023.3345800.
- Bass, L., Clements, P., Kazman, R., 2012. Software Architecture in Practice: Software Architect Practice_c3. Addison-Wesley.
- Becker, C., Chitchyan, R., Duboc, L., Easterbrook, S., Penzenstadler, B., Seyff, N., Venters, C.C., 2015. Sustainability design and software: The Karlskrona manifesto. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. Vol. 2, IEEE, pp. 467–476.
- Bennett, K.H., Rajlich, V.T., 2000. Software maintenance and evolution: a roadmap . In: Proceedings of the Conference on the Future of Software Engineering. pp. 73–87.
- Bennett, K.H., Ramage, M., Munro, M., 1999. Decision model for the legacy systems. In: IEE Proceedings-Software. Vol. 146, pp. 153–159.
- Bianchi, A., Caivano, D., Marengo, V., Visaggio, G., 2003. Iterative reengineering of legacy systems. IEEE Trans. Softw. Eng. 29 (3), 225–241.
- Candela, I., Bavota, G., Russo, B., Oliveto, R., 2016. Using cohesion and coupling for software re-modularization : Is it enough? ACM Trans. Softw. Eng. Methodol. (TOSEM) 25 (3), 1–28.
- Capra, E., Francalanci, C., Merlo, F., 2010. The economics of community open-source software projects: an empirical analysis of maintenance effort. Adv. Softw. Eng. 2010.
- Chidamber, S., Kemerer, C., 1994. A metrics suite for object-oriented design. IEEE Trans. Softw. Eng. 20 (6), 476–493. http://dx.doi.org/10.1109/32.295895. Curtis, B., Sappidi, J., Szynkarski, A., 2012. Estimating the size, cost, and type of technical debt. In: 2012 Third International Workshop on Managing Technical Debt. MTD, IEEE, pp. 49–53.
- Dams, D., Mooij, A., Kramer, P., Rădulescu, A., Vaňhara, J., 2018. Model-based software restructuring: Lessons from cleaning up COM interfaces in industrial legacy codes . In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering. SANER, pp. 552–556. http://dx.doi.org/10.1109/SANER.2018.8330258.
- https://www.digitalocean.com/community/conceptual-articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design
- https://stackoverflow.com/questions/30616660/which-solid-principles-are-violated
- Java-8 Documentation https://docs.oracle.com/javase/8/docs/api/java/util/Properties.html