NUANSA INFORMATIKA
Volume 20 Nomor 1, Januari 2026

p-ISSN :1858-3911* , *e-ISSN : 2614-5405
https://journal.fkom.uniku.ac.id/ilkom

A Study on the Application of the Iterator Pattern to Enhance Software Reusability

Siti Herawati [1],
Jl. HS.Ronggo Waluyo, Puseurjaya, Telukjambe Timur, Karawang, Jawa Barat 41361, Indonesia
Telp : (0826) 267641177
Email : 2210631170153@student.unsika.ac.id[1] ,

Abstract

The iterator design pattern is an effective approach in software development to achieve greater modularity and reusability. This study analyzes the application of the iterator pattern in enhancing software reusability, particularly within the object-oriented programming paradigm. The research includes a literature review and hands-on implementation using Python and Java to evaluate the separation of iteration logic from data structures. The results indicate that the pattern supports the principle of separation of concerns, enabling the development of code that is more flexible, structured, and maintainable. Code evaluation shows improvements in readability and the ease of component reuse. Therefore, this study provides a practical contribution to the understanding of design patterns and is expected to serve as a useful reference for software developers in building efficient and sustainable systems

Keywords*—Design Pattern, Iterator, Reusability, Object-Oriented Programming*

Submission: December 1, 2025 Accepted: December 22, 2025 Published: January 20, 2026
DOI Article: https://doi.org/10.25134/ilkom.v20i1.525

1. INTRODUCTION

Modern software systems are rapidly evolving in both scale and complexity. The demand for fast, stable, and efficient feature delivery encourages software engineering practices that emphasize modularity, separation of concerns, and the ability to reuse code components (reusability). Empirical studies indicate that improvements in reusability are strongly influenced by appropriate refactoring processes [1]. Other literature asserts that the internal quality of software can be enhanced through the selection of suitable design patterns, which support more consistent and maintainable code structures [2].

The iterator design pattern is a behavioral pattern that provides a standardized mechanism for traversing elements within a collection without exposing its internal structure. This pattern supports principles such as separation of concerns and loose coupling, both of which are fundamental to achieving flexible software architecture [3]. In addition, the iterator pattern has been proven to enhance the reusability of traversal logic across various programming contexts.

Refactoring plays a crucial role in increasing software reusability, and multiple studies show that effective refactoring can improve high-level design structures, including packages, classes, and methods [4]. Refactoring challenges often arise from the presence of code smells, which require structural fixes and consistent use of design patterns [5]. Studies on refactoring-related questions from developer communities also show that many developers struggle to determine optimal refactoring strategies, making design patterns such as the iterator a stable and reliable solution [6].

In modern system architectures, such as microservices, design patterns have a significant impact on system quality. Empirical studies show that well-designed patterns can balance quality trade-offs involving performance, modularity, and reusability [7]. Other research demonstrates that certain design patterns influence microservice performance, especially in large-scale scenarios [8]. Experimental evaluations further confirm that choosing the right architectural pattern can improve system performance without reducing structural flexibility [9].

However, the application of design patterns—including iterator—is not without challenges. Recent studies note that misuse or incorrect implementation of design patterns can lead to bad smells, especially in classical gang of four patterns [10]. Similar risks appear in object-oriented design models that fail to use patterns

p-ISSN :1858-3911* , *e-ISSN : 2614-5405
https://journal.fkom.uniku.ac.id/ilkom

properly, which can reduce reusability and maintainability [11]. Therefore, proper documentation, consistent implementation, and a deep understanding of the iterator pattern are essential before applying it in real development environments.

Various implementations of the iterator pattern have continued to evolve, including custom iterators, lazy iterators, generator-based iteration, and stream-based iteration commonly used in java, python, and c#. Research shows that pattern-based approaches improve development efficiency and enhance component reusability [12]. Studies on design patterns in microservices and devs modeling also confirm that design patterns contribute to building more modular and extensible components.

In the context of local software development in indonesia, several studies show that implementing design patterns can improve software quality and system maintainability. Research involving prototype-based user interface design for car rental applications demonstrates improved structural effectiveness in web-based systems [13]. The development of web-based financial applications also shows enhanced maintainability through modular design structures [14]. Similarly, studies on tourism information systems highlight how modular structures improve readability and component reusability [15].

2. LITERATURE REVIEW

2.1 Software Reusability

Reusability is the ability of a software module to be used again in other systems or projects without requiring significant modifications. Reusability can reduce development costs, accelerate production time, and improve software consistency and quality. Reusable components typically have a modular structure, clear interfaces, and well-written documentation. The application of design principles such as *low coupling* and *high cohesion* also supports the improvement of reusability.

2.2 Iterator Design Pattern

The iterator pattern is one of the behavioral design patterns used to access elements in a data collection sequentially without exposing its internal structure. This pattern separates the iteration logic from the data structure, thereby supporting flexibility and reusability. In its implementation, this pattern uses a special object (an iterator) that is responsible for the traversal process. It is commonly used in programming languages such as Java and Python and strongly supports the principle of separation of concerns.

3. METHODOLOGY

This method was chosen because it is suitable for exploring technical phenomena in depth through conceptual analysis, literature review, and exploratory code implementation. The research was carried out in several structured stages, including:

3.1 Literature Review

The study began with a literature review to understand design patterns, particularly the iterator pattern, and its contribution to reusability in software development. The primary reference examined was Design Patterns: Elements of Reusable Object-Oriented Software by [not in provided list], supported by scholarly journals, conference articles, and recent online sources. The literature also included previous studies that investigated the impact of design patterns on software quality aspects such as readability, modularity, and maintainability. This approach provided a solid theoretical foundation and contextual understanding of iterator pattern application.

3.2 Code Implementation and Analysis

After establishing a conceptual understanding through the literature, the research proceeded with the practical implementation of the iterator pattern in two widely-used object-oriented programming languages: Python and Java. These languages were chosen to demonstrate how the iterator pattern can be adapted to different paradigms while maintaining the same object-oriented foundation.

Two case studies were developed in this stage:

a) Case Study A (Python): A custom data structure was developed with iterator support using the __iter__() and __next__() methods, in accordance with Python's iterator protocol. This case focused on a simple collection such as a list of objects that required various traversal strategies.
b) Case Study B (Java): The Iterator and Iterable interfaces were implemented, and a custom iterator class was created to allow flexible traversal, including both forward and reverse iteration.

Each case study included code samples before and after applying the iterator pattern. The initial (baseline) code used a conventional iteration approach with traversal logic embedded directly in the data structure, while the refactored code

p-ISSN :1858-3911* , *e-ISSN : 2614-5405
https://journal.fkom.uniku.ac.id/ilkom

separated the iteration logic into a distinct iterator object. The analysis focused on code structure, responsibility grouping, level of coupling between classes, and the potential for code reuse in other application contexts.

3.3 Reusability Evaluation

The final stage involved evaluating the impact of applying the iterator pattern on software reusability. This evaluation was conducted qualitatively by comparing the code before and after refactoring based on several key reusability indicators:

a) Separation of Concerns: Assessing the extent to which iteration logic is separated from data and business logic.
b) Modularity: Evaluating the ability of the code to be divided into independent functional units.
c) Readability: Measuring improvements in code clarity and consistency following the implementation of the iterator pattern.
d) Reusability: Analyzing the potential for reusing the iterator components in other projects or with different data collections.

The evaluation also considered good software engineering practices, such as the open/closed principle, low coupling, and high cohesion. The findings from this evaluation were used to draw conclusions about the effectiveness of the iterator pattern in supporting the development of more structured, efficient, and maintainable software systems.

4 RESULTS AND DISCUSSION

4.1 Implementation Results

In object-oriented software development, managing and traversing data collections is a common task. Before applying the Iterator pattern, element traversal within a collection is typically performed manually using index-based loops, resulting in repetitive and poorly structured code. By implementing the Iterator pattern—both in Python and Java—data collection traversal can be performed in a more systematic way, separated from the logic of data storage. This approach enhances code readability and maintainability.

a. Example of Implementation Before Using Iterator in Python:

```
def print_books(self):
    print("=== Book List ===")
    for i in range(len(self.books)):
        print(f"{i+1}. {self.books[i]}")
```

Figure 1 : Before using iterators

The print_books(self) method is used to display the list of books stored in the collection in a structured format. This method prints the list title and then iterates through the collection using numeric indexing to display each book in sequence. This presentation makes it easier for users to verify the data and supports debugging and validation processes in text-based software development.

b. Example of Implementation After Using Iterator in Python:

```
class BookIterator:
    def __init__(self, books): self.books = books
self.idx = 0
  def __iter__(self):return self
  def __next__(self):
      if self.idx < len(self.books):
          b = self.books[self.idx]self.idx += 1; return
b
      raise StopIteration

class BookCollection:
    def __init__(self):self.books = []
    def                __iter__(self):              return
BookIterator(self.books)
```

Figure 2 : After using iterators

The code above defines an iterator for the BookCollection class. BookIterator allows the book collection to be accessed sequentially using a for loop without the need for manual indexing. By implementing the __iter__() and __next__() methods, the code simplifies the data traversal process and enhances modularity and readability when manipulating the book collection.

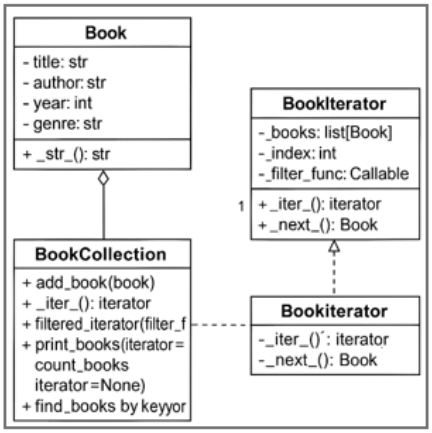


Figure 3 : Class diagram UML (Python)

The class diagram (Python) illustrates a book collection system. The Book class contains basic attributes of a book, such as title, author, year, and genre. BookIterator is a custom iterator with filtering capabilities, allowing flexible iteration over the book collection. BookCollection stores a list of books and provides various methods such as adding books, printing, filtering by genre/year, searching by keywords, and sorting by title.

c. Example of Implementation Before Using Iterator in Java:

p-ISSN :1858-3911* , *e-ISSN : 2614-5405
https://journal.fkom.uniku.ac.id/ilkom

```
public void printSongsForward() {
   System.out.println("=== Daftar Lagu ===");
   for (int i = 0; i < songs.size(); i++) {
        Song song = songs.get(i);
        System.out.println((i + 1) + ". " +
song.getTitle() +
        " - " + song.getArtist());
    }
}
```

Figure 4 : Before using iterators

The printSongsForward() method is used to display the list of songs in the Playlist object. By using index-based iteration, each song is printed sequentially along with its number, title, and artist name. This function provides a structured text representation of the playlist content, making it easier for users to view and verify the added songs.

d. Example of Implementation After Using Iterator in Java:

```
class SongIterator implements Iterator<Song> {
    private List<Song> songs; private int i = 0;
    public SongIterator(List<Song> s) {songs = s;
}
    public boolean hasNext() {return i <
songs.size(); }
    public Song next() { return songs.get(i++); }
}

class Playlist implements Iterable<Song> {
    private List<Song> songs = new
ArrayList<>();
    public Iterator<Song> iterator() {
       return new SongIterator(songs);
    }
}
```

Figure 5 : After using iterators

This code applies the Iterator pattern in Java to enable structured traversal of Playlist objects. SongIterator manages manual iteration over the list of songs, while Playlist provides iterative access to the song collection by implementing the Iterable interface. This approach improves modularity and flexibility in data management and separates the traversal logic from the data structure.

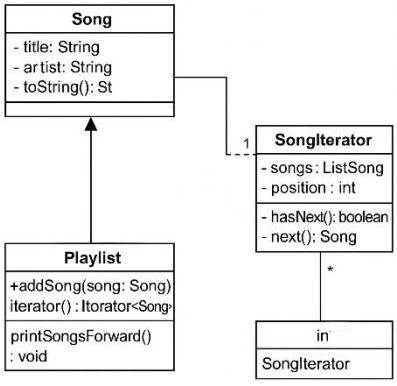


Figure 6 : Class diagram UML (Java)

The class diagram for the Java implementation illustrates a music playlist system. The Song class represents the song entity, containing attributes such as title and artist. SongIterator is a custom iterator that allows traversal of the song collection one item at a time, while the Playlist class stores the list of songs and provides methods for adding songs and printing the song list using the iterator.

4.2 Code Feature Comparison Table

The following table compares the coding approaches before and after the implementation of the iterator pattern and filtering functions in Python and Java. The focus is on aspects such as collection iteration, filtering capabilities, separation of concerns, and development flexibility. The new approach improves code reusability and facilitates the development of additional features.

Table 1 : Comparison of code features before and after applying Iterator

Fitur	Python (Before)	Python (After)	Java (Before)	Java (After)
Collection Iteration	for i in range(len(...))	for item in iterator using *BookIterator*	for (int i = 0; i < list.size(); i++)	for (Song s : this) using *Iterator*
Filtering and Searching	Hardcoded loops & conditions	Flexible filter functions through iterator	Hardcoded loops & conditions	Not available (basic iteration code)
Separated Iteration Logic	No	Yes, *BookIterator* is separated from the collection	No	Yes, *SongIterator* is separated
Reusability	Low (repetitive code)	High (modular, customizable)	Low	Medium (modular iteration, no filtering)
Separation of Concerns (SoC)	Unclear	Clear separation between iteration and	Unclear	Clearer with *Iterable*

p-ISSN :1858-3911* , *e-ISSN : 2614-5405
https://journal.fkom.uniku.ac.id/ilkom

		collection logic		
Extensibility	Limited (duplicate code)	High (easily extendable via new filters/iterators)	Limited	Medium

4.3 Reusability Evaluation

The implementation of the Iterator pattern in Python improves reusability, as filtering functions can be passed as parameters without modifying the BookCollection class. The iterator can be reused in various contexts with high flexibility. Meanwhile, Java also demonstrates improvement after using the Iterator pattern, although it remains less flexible than Python. For better results, Java can be combined with Predicate<Song> or Stream APIs to support more dynamic filtering capabilities

Table 2 : Evaluation of reusability in Python and Java

Fitur	Python (Before)	Python (After)	Java (Before)	Java (After)
Code Reusability	Low	High	Low	Medium
Ease of Modification	Low	High	Low	Medium
Extensibility Support	Limited	Flexible	Limited	Modular
Separation of Concerns	Weak	Strong	Weak	Good
Unit Test Support	Limited	Easy	Limited	Easy



Figure 7 : Bar chart of reusability evaluation

The figure above illustrates the reusability evaluation of the code before and after applying the Iterator pattern in both Python and Java. It shows that after adopting the Iterator, both languages experienced improvements across all evaluation aspects, such as code reuse, ease of modification, extensibility, separation of concerns, and support for unit testing. Python showed the most significant improvement, particularly in terms of modularity and separation of concerns. Java also improved, though its gains were more limited compared to Python. This demonstrates that the Iterator pattern helps enhance the structure and flexibility of code when managing data collections.

5 CONCLUSION

The application of the Iterator design pattern has proven to enhance the quality of reusability in object-oriented software development. By separating iteration logic from data structure, developers can produce code that is more modular, flexible, and easier to maintain. This implementation also aligns with core design principles such as separation of concerns and the open/closed principle.

Through case studies conducted in Python and Java, it was observed that Python offers greater flexibility due to its ability to accept filtering functions directly within the iterator. Meanwhile, Java also demonstrated improvements after implementing the Iterator pattern, although its filtering flexibility remains limited without additional constructs such as Predicate or Stream.

Overall, the use of the Iterator pattern not only simplifies the traversal of data collections but also strengthens the architecture of software systems to be more reusable and maintainable. Therefore, this pattern is highly recommended for projects that involve repeated and flexible data processing.

Furthermore, the Iterator pattern offers advantages in testing and continuous development, as its separation of iteration logic allows for more focused unit testing without dependency on the internal structure of the data. With clearer task isolation, development teams can accelerate refactoring processes and add new features without disrupting existing code. This makes the Iterator pattern a relevant and adaptive design solution for supporting modern software development practices that prioritize quality, efficiency, and scalability.

6 SUGGESTIONS

Based on the findings of this study, future work should extend beyond the Iterator pattern and investigate additional design patterns—such as Strategy, Observer, or Visitor—to provide a broader comparison of their impact on software

p-ISSN :1858-3911 **,** ***e-ISSN : 2614-5405***
https://journal.fkom.uniku.ac.id/ilkom

reusability. Moreover, applying the Iterator pattern within microservice architectures or other complex systems would help assess its effects on scalability and maintainability in real-world scenarios.

Subsequent research is also encouraged to incorporate quantitative metrics—such as coupling, cohesion, and the maintainability index—to support objective evaluation. Leveraging functional-programming features like filter() or Java Streams can further enhance the flexibility of iterators. By combining sound design practices with automated testing approaches, developers can build systems that are more modular, efficient, and sustainable over the long term.

REFERENCES

[1] E. A. Alomar, T. Wang, V. Raut, M. W. Mkaouer, C. Newman, and A. Ouni, "Refactoring for reuse: an empirical study," *Innov Syst Softw Eng*, vol. 18, no. 1, pp. 105–135, 2022.

[2] G. Vale, F. F. Correia, E. M. Guerra, T. O. Rosa, J. Fritzsch, and J. Bogner, "Designing Microservice Systems Using Patterns: An Empirical Study on Quality Trade-Offs," *IEEE* , pp. 69–79, 2022.

[3] R. Pinciroli, A. Aleti, and C. Trubiani, "Performance Modeling and Analysis of Design Patterns for Microservice Systems," *ICSA*, 2023.

[4] F. Wedyan and S. Abufakher, "Impact of Design Patterns on Software Quality: A Systematic Literature Review," *IET Software*, vol. 14, no. 1, pp. 1–17, 2020.

[5] G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Gueheneuc, "Code Smells and Refactoring: A Tertiary Systematic Review of Challenges and Observations," 2020.

[6] A. Peruma, S. Simmons, E. A. AlOmar, C. D. Newman, M. W. Mkaouer, and A. Ouni, "How Do I Refactor This? An Empirical Study on Refactoring Trends and Topics in Stack Overflow," 2021.

[7] S. H. S. Almadi, D. Hooshyar, and R. B. Ahmad, "Bad Smells of Gang of Four Design Patterns: A Decade Systematic Literature Review," *Sustainability*, vol. 13, no. 18, 2021.

[8] S. Ahmed, "Enhancing Software Development Efficiency: The Role of Design Patterns in Code Reusability and Flexibility," *AI Publications / IJEBDM*, vol. 19, no. 1, 2025.

[9] M. El Amine Hamri, "An Object-Oriented Framework for Designing Reusable and Maintainable DEVS Models using Design Patterns," 2020.

[10] W. Meijer, C. Trubiani, and A. Aleti, "Experimental evaluation of architectural software performance design patterns in microservices," *Journal of Systems and Software*, vol. 218, 2024.

[11] T. P. Y. Titan, Budiman, and J. H. F. E. Putra, "Perancangan Prototype User Interface Dan Pengujian User Experience Aplikasi Rental Mobil Berbasis Menggunakan Metode Design Thinking (Studi Kasus : Pt Trans Berjaya Khatulistiwa)," *Nuansa Informatika*, vol. 17, no. 2, 2023.

[12] Darsiti and D. S. F. Lestari, "Pengembangan Aplikasi Keuangan Berbasis Web (Studi Kasus : Mts AT-Taufiq Kota Bandung)," *Nuansa Informatika*, vol. 17, no. 2, 2023.

[13] M. K. Maulidani, Darsanto, and M. E. Iswanto, "Rancang Bangun Aplikasi Informasi Wisata Kabupaten Indramayu Berbasis Mobile Android," *NUANSA INFORMATIKA*, vol. 18, no. 2, 2024.

[14] A. Kevin and H. C. Kurniawan, "Analisis Pengaruh Design Pattern Terhadap Pemeliharaan Perangkat Lunak Learning Management System," *Jurnal Telematika*, vol. 18, no. 1, 2023.

[15] I. G. B. V. Putraadinatha, D. D. J. Suwawi, and S. Y. Puspitasari, "Pengaruh Design Pattern Terhadap Maintainability Aplikasi Mobile," *eProceedings of Engineering*, vol. .8, no. 5, 2021.