Enhancing Undergraduate Proficiency Through the Iterator Pattern in Programming Pedagogy

MARY TEDESCHI[1], Pace University
DEEP MEHTA[2], Pace University
PULKIT SINGH[3], Pace University
SEJAL ARORA[4], Pace University
ARPIT SHAH[5], Pace University
NIHAL KANIYERI[6], Pace University
CHINTAN CHAUHAN[7], Pace University
SERGIO BELICH[8], NYC College of Technology

Introducing difficult topics first builds a strong foundation for tackling complex concepts. However, teaching introductory programming remains a significant issue as students struggle to link theory with practice. This study investigates the ability of undergraduate students to implement the Iterator pattern, a fundamental behavioral design pattern, across various programming languages using online coding platforms. Introducing abstract concepts early lets students channel their enthusiasm into mastering advanced material. This new knowledge then makes it easier to grasp simpler topics that build upon it. Patterns in education of computer science still need more thorough research. The research will introduce the Iterator pattern early in the course curriculum, assessing student comprehension at multiple stages. Structured lectures, coding examples, hands-on projects, and formative assessments on the Integrated Development Environment (IDE) are used to evaluate students' understanding. Quantitative performance data and qualitative feedback highlight common recurring issues, including syntactical misunderstandings and logical errors. Use of instructional strategies—such as: hands-on practice, pair programming, visual aids, worked examples, and structured support, when combined greatly improves student proficiency. By identifying effective pedagogical approaches, optimal curriculum sequencing, and robust support techniques, the research aims to ensure students develop an enhanced understanding of design patterns and iterative solutions. This foundational knowledge is imperative for efficient and optimized software development, essential for their future careers as software engineers or computer scientists. The findings provide valuable insights for curriculum design, assisting educators in prioritizing and sequencing programming concepts while offering targeted support to novice programmers.

Categories and Subject Descriptors: K.3.2 [**Computer and Information Science Education**] Information systems education

General Terms: Computers and Education

Additional Key Words and Phrases: Conceptual understanding, design patterns, instructional strategies, iterator pattern, teaching methods

ACM Reference Format:

Tedeschi, M. et al. 2024. Enhancing Undergraduate Proficiency Through the Iterator Pattern in Programming Pedagogy. jn V (October 2024), 14 pages.

Author's address: M. Tedeschi, Pace University, 15 Beekman Street, NY NY 10038; email:mtedeschi@pace.edu; D. Mehta, Pace University, 15 Beekman Street, NY NY 10038; email:dm29655n@pace.edu; P. Singh, Pace University, 15 Beekman Street, NY NY 10038; email:ps79178n@pace.edu; S. Arora, Pace University, 15 Beekman Street, NY NY 10038; email:sarora@pace.edu; A. Shah, Pace University, 15 Beekman Street, NY NY 10038; email:as61268n@pace.edu; N. Kaniyeri, Pace University, 15 Beekman Street, NY NY 10038; email:nk43213n@pace.edu; C. Chauhan, Pace University, 15 Beekman Street, NY NY 10038; email:cc95065n@pace.edu; S. Belich, NYC College of Technology, 300 Jay Street, Brooklyn, NY 11201; email:Serge.Belich85@Citytech.Cuny.Edu;

1. INTRODUCTION

By learning the iterator pattern, you gain the ability to write efficient, maintainable, and scalable code that adheres to best practices in software design. First-year college students are still having difficulty understanding the general concept of a loop in our previous semester. A loop is a series of statements in a computer program that are to be executed repeatedly. Teaching programming languages effectively is fundamental to computer science education. Mastering iterative patterns, such as for loops, while loops, recursion, and the use of libraries across different programming languages, is crucial for automating tasks and implementing complex algorithms [1; 2]. The Iterator pattern is a design pattern that provides a way to access elements of an aggregate object sequentially without exposing its underlying representation. This pattern is commonly used in object-oriented programming and plays a crucial role in the design of collections and aggregate structures flexibly and reliably [8; 9]. There are multiple ways to teach loops, and the iterator pattern is just one of them. The iterator pattern focuses on "what" needs to be done (traversal) rather than "how" (managing indices or internal collection logic). This abstraction makes it an ideal teaching tool for introducing loops without overwhelming learners with low-level details.

This study focuses on assessing undergraduate students' ability to implement the Iterator pattern across multiple programming languages. By immersing students in various iterative constructs and requiring them to apply the pattern to solve problems and complete projects, our aim is to observe their learning trajectory, identify common obstacles, and evaluate the effects of alternative teaching methodologies. Patterns like the iterator pattern aren't teaching methods, but they can enrich teaching by providing structured, real-world examples of programming concepts. The real "alternative methods" lie in how those patterns are introduced, such as through hands-on coding, peer discussions, or interactive demos. Ultimately, this research seeks to develop strategies to improve programming education by improving students' knowledge in iterative patterns.

To establish a baseline understanding of students' initial knowledge and skills, led by a computer science professor, they will complete a practical assignment during the first month of class. By engaging students in hands-on learning experiences, the study aims to assess their ability to apply the Iterator pattern effectively across different programming contexts. Metrics are typically used as tools to quantify performance, progress, or quality. In this case we are looking at learning outcomes such as metric examples, the number of correct solutions to exercises involving the iterator pattern. Just how many of the students actually completed the assignment in either Java or Python. The purpose is to gauges whether students understand and can apply the concept. The course curriculum did immerse students in a programming language. Ultimately, the goal is to pave the way for the development of strategies that enhance programming education.

This paper discusses the results of the initial assignment through a meticulous analysis of student performance data and feedback. In addition, the study will explore the impact of instructional strategies such as hands-on practice, peer learning, and visual aids, evaluating their effectiveness in enhancing student understanding and proficiency.

This investigation aims to bridge the gap in undergraduate proficiency in iterative patterns across programming languages. The goal is to contribute to the development of more effective pedagogical strategies that can better prepare students for their future careers in computer science. Through comprehensive analysis, this research aspires to enhance programming education by leveraging the Iterator pattern to promote student knowledge in iterative constructs.

The structure of this paper includes an exploration of the study's motivation, a concept overview, detailed methods, insights from the student perspective, an overview of the Iterator pattern, and a discussion of the recent study's results followed by the conclusion.

2. MOTIVATION FOR THIS STUDY

Enhancing the effectiveness of introductory programming courses is a significant goal in the realm of computer science education. These courses serve as the foundational building blocks for students embarking on their journey into the world of programming and software development. However, traditional teaching methodologies often fail to address key challenges faced by students, such as syntax barriers and inadequate problem-solving skills.

In "Understanding the Syntax Barrier for Novices" (2011) by Paul Denny et al., [12] the authors highlight the difficulties students encounter in connecting theoretical concepts with practical applications, frequently resulting in syntax errors that prevent meaningful feedback on their logic. This barrier is significant, as it impedes the learning process and underscores the need for revised teaching strategies. The study emphasizes the necessity for changes in teaching methodologies to overcome these barriers and improve the practical application of theoretical knowledge.

Similarly, the IEEE and ACM Joint Task Force on Computing Curricula (2001) [13] identified problem solving as a critical area of weakness among students. The author states that the primary objective of an introductory programming course is to cultivate students' problem-solving abilities and introduce fundamental design and programming concepts. In "Teaching Introductory Programming: A Quantitative Evaluation of Different Approaches" (2015) [14], Theodora Koulouri et al. discuss the complexities of teaching programming, highlighting misunderstandings in problem decomposition and fundamental principles. The use of Java as an introductory language, with its object-oriented focus, can distract from basic programming concepts due to its syntax complexity [11]. Therefore, introducing the Iterator Pattern early can serve as an entry point to these object-oriented concepts, facilitating a smoother transition for students by focusing on iteration as a common thread across languages.

Irit Hadar and Ethan Hadar, in "An Iterative Methodology for Teaching Object-Oriented Concepts" (2007) [15], emphasize the importance of abstract thinking, proposing iterative methodologies to integrate modeling and coding activities. Similarly, the Iterator Pattern encourages abstract thinking, allowing students to focus on problem solving without being bogged down by syntax and implementation details.

Franklyn Turbak et al. in "Teaching Recursion before Loops in CS1" (1999) [16], also advocate for reversing the traditional sequence of teaching recursion and loops to aid understanding [4]. This insight parallels the idea of introducing design patterns such as the Iterator pattern early to establish a conceptual framework that enhances comprehension of iterative and recursive constructs.

Therefore, by integrating the Iterator pattern earlier in introductory programming courses, we aim to address the syntax barriers and problem-solving deficiencies that plague novice programmers. This approach not only bridges the gap between theory and practice, but also equips students with the tools necessary for efficient and effective programming. Through this study, we seek to explore and validate innovative pedagogical strategies that can transform programming education and better prepare students for future careers in computer science.

TERMINOLOGY

The terms "iterator" and "iterative" are related but distinct and are often used in different contexts within computer science and programming.

Iterator

An iterator is an object that enables traversal over elements of a collection (such as an array or a list). It provides a way to access elements sequentially without exposing the underlying structure. The key characteristics of an iterator include:

Methods: Typically, an iterator implements methods like next() and hasNext() (or similar), which allow moving to the next element and checking if more elements are available. Usage: Iterators are used in loops to traverse collections, enabling the programmer to process each element in turn. Examples: In Python, an iterator can be created using iter() and elements can be accessed using next(). In Java, the Iterator interface provides hasNext() and next() methods.

Iterative

The term iterative refers to a process or methodology that involves repetition. It describes a technique that repeatedly applies a certain set of steps to approach a desired goal or solution. Key characteristics of iterative processes include:

Repetition: Involves repeatedly executing a block of code until a certain condition is met. Usage: Used in loops (like for loops and while loops) and in algorithms that require multiple passes or updates (like iterative refinement)[1]. Examples: Iterative algorithms include techniques like gradient descent in machine learning, where the algorithm iteratively adjusts parameters to minimize a cost function.

Summary of Differences

Iterator: A specific object or interface for traversing elements in a collection. Iterative: A general term describing a repetitive process or method. See Fig. 1 below for the example.

Example in Python

```
Iterator Example
# Creating an iterator
my_list = [1, 2, 3, 4]
my_iterator = iter(my_list)

# Using the iterator
print(next(my_iterator))  # Outputs: 1
print(next(my_iterator))  # Outputs: 2
Iterative Example

# Using an iterative process with a loop
for i in range(5):
    print(i)  # Outputs: 0, 1, 2, 3, 4
```

Fig. 1: Example 1

3. CONCEPT OVERVIEW

Iteration is the process of repeating a particular action. The keywords used could be for, while, or do-while. A for loop can be used either to obtain repetition or to make the variable available for some purpose. The do keyword is used to execute over and over until a condition becomes false, checking the condition at the end instead of the beginning, so that the loop is guaranteed to execute at least once. The introduction of the Iterator pattern marked a pivotal moment in the evolution of software engineering, revolutionizing how programmers traverse and access elements within collections. Before formalization of this pattern, software engineering grappled with less abstract and tightly coupled methods for iteration. The absence of a standardized approach led to increased complexity and reduced code re-usability, particularly in procedural programming environments. Christopher Alexander defines the term pattern as follows: Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution [17]. The students are given the Iterator pattern to learn a valuable teaching tool which is used in real-world software development and is used in many programming languages (such as Java, Python, and C#). Using existing patterns, students build on the collective experience of skilled software engineers. Every pattern deals with a specific, recurring problem in the design or implementation of a software system. When an expert works on a particular problem, it is unusual for them to tackle it by inventing a new solution that is completely different and distinct from existing ones. They often recall a similar problem that they have already solved and reuse the essence of the solution to solve the new problem. What Makes a Pattern? A three-part schema that underlies every pattern:

Context: Design situation giving rise to a design problem

Problem: Set of forces repeatedly arising in the context

Solution: A proven solution to the problem

- Configuration to balance the forces
    - Structure with components and relationships
    - Run-time behavior

Students will use this as a template and no two implementations of a given pattern are likely to be the same.

~~~
**While Loop**: It is like making pancakes until you run out of batter. You say, "
   Keep making pancakes while there is still batter left."
```python

batter_left = True

while batter_left:
    print("Making another pancake")
    # At some point, you set batter_left to False to stop the loop
```
~~~

Fig. 2: Example 2

Fig. 2 shows pseudocode explaining the concept. Historically, programmers embedded ad hoc traversal code directly within the collection's implementation, leading to drawbacks such as violating the Single Responsibility Principle, code duplication, and lack of reusability [8; 18]. To address these issues, the Iterator pattern emerged, promoting the separation of concerns, encapsulation, and re-usability [6; 7]. The history of the Iterator pattern dates back to the early days of programming, marked by ad hoc traversal methods embedded within collection implementations. Key milestones include the introduction of enumerators in languages such as Smalltalk [8]. However, it was formalized in the 1990s through the C++ Standard Template Library (STL) [10] and the work of the Gang of Four [23], revolutionizing iterative programming [6; 8]. This standardized, abstract approach promoted cleaner, more maintainable, and reusable code, with subsequent adoption in languages like Java and Python cementing its significance in modern software development [9]. The illustration in Table 1 shows the historical progression and development of the Iterator pattern in the context of software engineering.

In their book "Design Patterns: Elements of Reusable Object-Oriented Software" (1994), Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides introduced the Iterator pattern as one of the 23 essential design patterns. The Iterator pattern comprises several key concepts: the Iterator, which facilitates traversal through collection elements by providing methods for accessing, checking, and optionally removing elements; the Iterable, an object that can be iterated on, typically the collection itself, which offers an iterator() method to retrieve an iterator; and the principle of Separation of Concerns, which ensures that traversal responsibility is distinct from the collection [8; 9].

Teaching the Iterator pattern early in programming courses, we believe, can significantly influence students' coding practices, helping them avoid inefficient methods, and fostering a deeper understanding of code abstraction and reusability. A common exercise tasked students with printing the message "Hi Mom" multiple times. Without early exposure to iteration, students often resorted to a repetitive, manual approach, as shown in Example 1:

# Without Iteration

```
# Number of times to print "Hi Mom"
n = 5
print("Hi Mom")
print("Hi Mom")
print("Hi Mom")
print("Hi Mom")
print("Hi Mom")
```

Fig. 3: Example 3

This method, while straightforward, resulted in repetitive and inefficient code, Fig. 3, Example 3. Such practices can lead students to develop poor habits, as they manually duplicate code for each repetition. This makes the code cumbersome to modify, introduces possible errors, and reduces scalability. By introducing the Iterator pattern early in the course, students can be guided to adopt more efficient coding practices from the outset, as demonstrated in Fig. 4 Example 4:

# With Iteration

Teaching the Iterator pattern early helps students develop good coding habits and prevents them from adopting inefficient practices. This foundational knowledge prepares them for more advanced programming concepts, emphasizing the importance of design patterns in writing robust, maintainable software.

```python
# Number of times to print "Hi Mom"
n = 5
for _ in range(n):
    print("Hi Mom")
```

Fig. 4: Example 4

The Iterator pattern has evolved significantly throughout the history of computer science, with various implementations and improvements across different programming paradigms and languages. The following table presents a chronological overview of its development.

4. COURSE DETAILS AND LEARNING METHODS

There are two courses, one in Java and one in Python. Java is a classic object-oriented language, making it ideal for teaching the principles of encapsulation and inheritance that underpin the iterator pattern. Python supports multiple programming paradigms, allowing students to see how iterators can be implemented in a more dynamic and less rigid context.

5. COURSE DESCRIPTION

We have two different courses in two different universities.

5.1 Course 1:

This course focuses on developing solutions to programming problems using object-oriented techniques with the Python language. Students learn the fundamental elements of object-oriented programming, including the use of classes, objects, and Python libraries such as NumPy. The course introduces data structures and the use of UML (Unified Modeling Language) to emphasize programming problem solving. Evaluation in this course typically involves a combination of programming assignments, quizzes, and exams.

5.2 Course 2:

This two-semester course introduces fundamental concepts of computers and computing skills using the Java programming language. It covers foundational components of computer science, such as hardware and software components, algorithms, data types, loops, types and variables, objects, classes, methods, parameters, and object-oriented thinking. Chapters 1 to 6 provide a comprehensive foundation that enables students to analyze complex computing problems, design, implement, and evaluate computing-based solutions. Students are evaluated through a mixture of programming assignments, lab exercises, and exams.

In both courses, programming problem solving is emphasized throughout, and students engage in iterative exercises that progressively build their mastery in object-oriented programming. The combination of theoretical concepts and practical assignments ensures that students develop a robust understanding of programming principles and their applications.

6. METHODS AND STUDENT PERSPECTIVE

Students from Pace University will be assigned Python, while students from St. John's University will be designated Java, enabling a comparative analysis of learning outcomes. Pre-configured development environments will be provided to minimize setup time. For IDEs, Python students will use Pearson Revel, and Java students will use

Zybooks, ensuring a focus on learning rather than setup. [19] [20]

In the first week, students will be introduced to programming patterns, focusing on the Iterator pattern. They will complete a pre-test questionnaire to assess their prior knowledge and experience with programming and iteration concepts. The pre-test will include:

- What is your major?
- Have you taken any computer science or programming courses before? (Yes/No)
- Programming Experience: (1-5 scale from ‘None’ to ‘Expert’)
- Have you ever heard of the term ‘iteration’ in a general context? (Scale from ‘Never heard of it’ to ‘I am an expert’)
- Do you know what a ‘loop’ means in everyday language? (Scale from 1 to 5)
- Do you understand the concept of repeating a set of instructions multiple times to achieve a result? (Yes/No)
- Which of the following best describes a repetitive task you might perform in a non-programming context?
  - Reading a book
  - Brushing your teeth every morning and night
  - Writing a single email
- Which of these is an example of a repetitive action in using software?
  - Sending an email
  - Formatting a document
  - Applying the same format to multiple cells in a spreadsheet
- What do you think a computer program does when it ‘loops’ through a set of instructions?

Students will have four weeks to complete an assignment on implementing the Iterator pattern in Python and Java. Weekly check-ins and progress reports will monitor their development and provide feedback. The students will present their work at the end of the four weeks, discussing their implementation and solutions.

Following the presentations, a post-test questionnaire will evaluate their comprehension and application of iterative patterns. The post-test will include similar questions to the pre-test to measure improvements and additional questions to gather insights into their learning experiences. Quantitative data from assessments, pre-test and post-test responses will be statistically analyzed. Qualitative data from feedback, interviews, and presentation observations will be analyzed thematically.

Timeline and Milestones

The study will span four weeks, with key milestones including the introduction and distribution of assignments in Week 1, weekly check-ins and progress reports from Weeks 2 to 4, and final presentations and post-test evaluations in Week 4. Progress will be tracked through weekly check-ins and detailed progress reports to ensure that students receive timely feedback and support throughout the study.

Collection and Analysis of Student Questionnaires

Student questionnaires will be administered online before the first class (pre-test) and after the final presentations (post-test). Quantitative data from the questionnaires will be statistically analyzed to measure knowledge improvements, while qualitative data from feedback and observations during presentations will be thematically analyzed to provide deeper insights into the students’ learning experiences.

Evaluation of Student Work

Student work will be evaluated on the basis of several criteria to connect the methodology to the research statement on how the Iterator pattern can improve learning programming. The evaluation will include:

Correctness: Ensuring the code accurately implements the Iterator pattern. Efficiency: Assessing the performance and resource utilization of the implemented pattern. Readability: Evaluating the clarity and organization of the code, including appropriate use of comments and naming conventions. Application of Concepts: Measure how well students apply theoretical concepts to practical tasks. Problem-Solving Approach: Review of the steps taken to address and solve programming problems.

To facilitate this evaluation, the following can be used: an automated grading tool such as CodeGrade (from Pearson Revel), Zybooks, and/or a proprietary compiler. These tools will automatically check for correctness, efficiency, and basic readability. In addition, manual reviews will be conducted to assess the deeper aspects of code quality and problem-solving approaches. Detailed rubrics (Appendix A) will provide consistent and objective evaluations on all student submissions.

Using this comprehensive evaluation strategy, the study aims to provide concrete evidence of the effectiveness of teaching the Iterator pattern in improving student programming skills. The evaluation criteria and tools will be detailed in an Appendix to ensure transparency and reproducibility of the study's findings.

Planned Evaluation

The results of this study will be analyzed to identify common recurring issues, evaluate the value of distinct teaching strategies, and assess the student's aptitude to implement the Iterator pattern.

Efficacy of Teaching Strategies

The effectiveness of different teaching strategies will be evaluated based on student performance and feedback, once available. The following strategies are being considered for evaluation:

Structured Lectures: In-depth explanations and visual aids during lectures.

Coding Examples: Practical coding examples that demonstrate the Iterator pattern in different programming languages. Hands-On Project: The project was designed to reinforce the practical application of the Iterator pattern. Pair Programming: Collaborative exercises to enhance understanding through peer interaction. Structured Support: Regular support sessions, including office hours and discussion forums.

Student Proficiency

We plan to assess student competency in implementing the Iterator pattern by evaluating:

Pattern Implementation: Ability to correctly implement the Iterator pattern in various programming languages. Scenario Application: Skill in applying the pattern to different problem-solving scenarios. Understanding Components: Comprehension of the pattern's components and their interactions.

Curriculum Design

Insights from the study will inform the design of programming curricula, specifically:

Prioritization and Sequencing: Adjustments in the order in which programming concepts are introduced. Hands-On Project: Introduction of a practical project in the first week, to reinforce learning from the outset.

Discussion

Does it matter if students are using Python or Java in our classes to learn the Iterator pattern? What about the IDE?

The field of computer science education is undergoing rapid transformation, and educators are actively seeking innovative strategies to bridge the gap between theoretical knowledge and practical application in introductory programming courses. Recent research has delved into various approaches to enhance student proficiency, with a particular emphasis on design patterns, interactive learning environments, and formative assessment techniques.

Studies have illuminated the possibilities of incorporating design patterns, such as the Iterator pattern, into the curriculum to foster code reusability, maintainability, and a deeper comprehension of software design principles. By introducing students to established patterns early on, educators can equip them with a valuable toolkit to tackle complex programming challenges and develop robust and efficient code. For example, understanding the Iterator pattern can enable students to create flexible and adaptable code structures, leading to improved problem-solving abilities[3].

Moreover, the integration of online coding platforms and interactive development environments has emerged as a powerful tool for enhancing student engagement and learning outcomes. These platforms provide opportunities for hands-on practice, immediate feedback, and collaborative learning, thus fostering a deeper understanding of programming concepts. Using interactive exercises and simulations, educators can create dynamic and engaging learning experiences that cater to diverse learning styles.

Equally crucial is the role of formative assessment in the learning process. Regular evaluation allows educators to identify student misconceptions promptly, provide targeted feedback, and adjust instructional strategies accordingly. By incorporating formative assessment techniques, such as quizzes, coding challenges, and peer reviews, instructors can monitor student progress, identify areas of difficulty, and offer timely support.

Understanding the relationship between contextual factors, such as problem formulation or student background, that relate to performance on iteration and recursion problems can help inform pedagogy [5]. Building on these established research findings, this study investigates the advantage of introducing the Iterator pattern early in the curriculum through the use of online coding platforms and a comprehensive assessment strategy. By examining student performance and gathering qualitative feedback, the research aims to contribute to the development of evidence-based pedagogical approaches that can be widely implemented in introductory programming courses. The findings of this study have the potential to inform instructional design, curriculum development, and the creation of effective learning materials, ultimately leading to improved student outcomes in computer science education.

For example, if the study demonstrates the effectiveness of early Iterator pattern introduction, educators can integrate pattern-based problem solving exercises into their courses, encouraging students to apply the pattern in various contexts. Similarly, if the results highlight the benefits of interactive learning environments, instructors can incorporate more online coding activities and simulations into their teaching, fostering active learning and experimentation. By understanding the impact of formative evaluation, educators can refine their assessment practices to provide more meaningful feedback and support to students [21; 22].

Conclusion

This research aims to enhance undergraduate proficiency in programming by investigating the implementation of the Iterator pattern across multiple programming languages. By introducing the Iterator pattern early in the course curriculum and employing various teaching strategies, the study aims to address key issues faced by students, such as syntax barriers and inadequate problem-solving skills. Although student feedback is not yet available, the findings will provide valuable insights for curriculum design, helping educators prioritize and sequence programming concepts while offering targeted support to novice programmers. Ultimately, the goal is to ensure that students develop a better understanding of design patterns and iterative solutions, equipping them with the skills needed for efficient and optimized software development in their future classes.

ACKNOWLEDGMENTS

The authors thank our shepherd, Michael Sale, for his detailed feedback on the improvements. We also thank the students who took the courses and provided the results in the survey. Special thanks to Richard Karcher for reviewing this piece and for helpful corrections. Thanks to Pace University for funding this research. Thanks to Cathy Dwyer for helpful discussions in developing the inital iterator pattern idea. Thanks to the peer review team: Christian Kohls, Dennis Dubbert, Anja Bertels, Charles Danoff, Sridevi Ayloo, Hiroaki Tanaka.

APPENDIX

Table I. : Historical progression & development of the Iterator pattern

Year	**Event**	**Description**
1960s	Introduction of Early Programming Constructs	Early programming languages like ALGOL and Lisp introduced basic iteration constructs such as loops (e.g., for, while), enabling repetitive execution of code.
1970s	Advent of Object-Oriented Programming and Enumerators	Object-oriented languages like Smalltalk introduced enumerators, allowing element-by-element access within collections, though traversal logic was closely tied to collections.
1983	Introduction of the “Collection” Class in Smalltalk-80	Smalltalk-80 introduced the Collection class with iteration methods, formalizing iteration in OOP.
1987	Publication of “Object-Oriented Software Construction” by Bertrand Meyer	Bertrand Meyer emphasized encapsulation and modularity, influencing iteration pattern development.
1990	Iterators Proposed for the C++ Standard Library	The inclusion of iterators was proposed for the C++ Standard Library, standardizing iteration in C++.
1990s	Standardization with the C++ Standard Template Library (STL)	The STL in C++ provided a consistent interface for accessing collection elements, promoting code reuse and modularity.
1994	Introduction of the Iterator Design Pattern	The “Gang of Four” identified the Iterator pattern [23], providing a structured approach to common software design problems.
1995	Java 1.0 Release with Iterator Interface	Java 1.0 included an Iterator interface, integrating the pattern into its core API.
2002	Introduction of Enhanced for Loop in Java	Java 5 introduced the enhanced for loop, simplifying iterator usage in client code.
2008	Python’s Iterators and Generators	Python 2.2 introduced iterators and generators, providing built-in support for the Iterator pattern.

Table II. : Preassessment Results

Criteria	**Excellent**	**Good**	**Satisfactory**	**Needs Improvement**
Percentage	Points per Grade: (25-23)	Points per Grade: (22-20)	Points per Grade: (19-17)	Points per Grade: (16-0)
Understanding of the Iterator Pattern	Clearly explains the concept of the Iterator Pattern. Identifies its purpose and benefits in software design. Uses appropriate terminology accurately.	Explains the concept of the Iterator Pattern with minor inaccuracies. Identifies its purpose and benefits with some gaps. Uses mostly appropriate terminology.	Provides a basic explanation of the Iterator Pattern. Identifies its purpose but with significant gaps or misunderstandings. Uses some appropriate terminology but with frequent errors.	Fails to explain the concept of the Iterator Pattern adequately. Misunderstands its purpose and benefits. Uses incorrect terminology or fails to use relevant terms.
Implementation of the Iterator Pattern	Correctly implements the Iterator Pattern in code. Code is clean, well-organized, and follows best practices. Provides clear and effective documentation/comments.	Code is mostly clean and organized, following most best practices. Provides documentation/comments that are mostly clear.	Implements the Iterator Pattern with several errors. Code is functional but lacks organization and may not follow best practices. Documentation/comments are present but unclear or incomplete.	Fails to correctly implement the Iterator Pattern. Code is poorly organized and does not follow best practices. Documentation/comments are missing or unclear.
Application of the Iterator Pattern	Demonstrates a thorough understanding of when and how to use the Iterator Pattern. Applies the pattern effectively in a relevant context or project. Shows creativity and insight in application.	Demonstrates a good understanding of when and how to use the Iterator Pattern. Applies the pattern appropriately in a relevant context. Shows some creativity and insight in application.	Demonstrates a basic understanding of when and how to use the Iterator Pattern. Applies the pattern in a relevant context but with some issues. Shows limited creativity and insight in application.	Fails to demonstrate an understanding of when and how to use the Iterator Pattern. Misapplies the pattern or uses it in an irrelevant context. Shows no creativity or insight in application.
Testing and Debugging	Thoroughly tests the implementation with a variety of test cases. Identifies and fixes all major and minor bugs. Demonstrates effective use of debugging tools and techniques.	Tests the implementation with several test cases, missing some edge cases. Identifies and fixes most bugs. Demonstrates good use of debugging tools and techniques.	Tests the implementation with basic test cases, missing many edge cases. Identifies and fixes some bugs but leaves several unresolved. Demonstrates basic use of debugging tools and techniques.	Fails to adequately test the implementation. Leaves many bugs unresolved. Demonstrates poor use of debugging tools and techniques.
Bonus Points: Presentation and Explanation	Presents the project clearly and confidently. Provides thorough explanations of choices and implementations. Answers questions accurately and insightfully. (5)	Presents the project clearly with minor issues. Provides good explanations of choices and implementations. Answers questions accurately but with some gaps. (4)	Presents the project with some clarity issues. Provides basic explanations of choices and implementations. Answers questions but with noticeable gaps. (3)	Struggles to present the project clearly. Provides unclear or incomplete explanations of choices and implementations. Struggles to answer questions accurately. (2-0)
Potential Total Points: Without Bonus: 100, With Bonus: 105				

REFERENCES

[1] Alan C. Benander, Barbara A. Benander, and H. Pu. Recursion vs. Iteration: An Empirical Study of Comprehension. *Journal of Systems and Software* 32, 1 (1996), 73–82.

[2] Jie Chao, David F. Feldon, and James P. Cohoon. Dynamic Mental Model Construction: A Knowledge in Pieces-Based Explanation for Computing Students' Erratic Performance on Recursion. *Journal of the Learning Sciences* 27, 3 (2018), 431–473. `https://doi.org/10.1080/10508406.2017.1392309`

[3] Nell B. Dale. Most difficult topics in CS1: results of an online survey of educators. *SIGCSE Bulletin* 38, 2 (2006), 49–53.

[4] Stephen Davies, Jennifer A. Polack-Wahl, and Karen Anewalt. A Snapshot of Current Practices in Teaching the Introductory Programming Sequence. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, Association for Computing Machinery, New York, NY, USA, 625–630. `https://doi.org/10.1145/1953163.1953339`

[5] Esteero, Ramy, Mohammed Khan, Mohamed Mohamed, Larry Yueli Zhang, and Daniel Zingaro. Recursion or Iteration: Does it Matter What Students Choose? *SIGCSE*, ACM, 2018, 1011–1016.

[6] Refactoring.Guru. Iterator. Last Accessed: 2024, August 5. `https://refactoring.guru/design-patterns/iterator`

[7] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. *Pattern-Oriented Software Architecture, Volume 1, A System of Patterns*. Wiley, 1996.

[8] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[9] Bloch, J. *Effective Java* (2nd ed.). Addison-Wesley, 2008.

[10] Meyers, S. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Addison-Wesley, 2001.

[11] Liskov, B., & Guttag, J. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.

[12] P. Denny, A. Luxton-Reilly, E. Tempero, and J. Hendrickx Understanding the syntax barrier for novices. *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education - ITiCSE '11*, 2011. `https://doi.org/10.1145/1999747.1999807`

[13] The Joint Task Force on Computing Curricula IEEE Computer Society Association for Computing Machinery *Computing Curricula 2001*. ACM/IEEE-CS Joint Task Force on Computing Curricula, 2001. `https://www.acm.org/binaries/content/assets/education/curricula-recommendations/cc2001.pdf`

[14] T. Koulouri, S. Lauria, and R. D. Macredie Teaching Introductory Programming. *ACM Transactions on Computing Education* 14, 4 (2015), 1–28. `https://doi.org/10.1145/2662412`

[15] Irit Hadar and Ethan Hadar An Iterative Methodology for Teaching Object Oriented Concepts. *Informatics in Education - An International Journal* 6, 1 (2007), 67–80. `https://www.ceeol.com/search/article-detail?id=946487`

[16] F. Turbak, C. Royden, J. Stephan, and J. Herbst Teaching Recursion Before Loops in CS1. *Journal of Computing in Small Colleges* 14, 4 (1999), 86–101. `https://cs.wellesley.edu/~fturbak/pubs/jcsc99.pdf`

[17] Christopher Alexander *The Timeless Way of Building*. Oxford University Press, 1979.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[19] C. Watson and F. W. Li The effectiveness of using online coding platforms to teach introductory programming courses. In *Proceedings of the 45th ACM technical symposium on computer science education*, 2014.

[20] C. Parnin, M. Hicks, L. Singer, and V. Barr Learning to code by watching others: A study on the effectiveness of online video tutorials. In *Proceedings of the 2013 ACM international symposium on professional and amateur software development*, 2013.

[21] P. Black and D. Wiliam Assessment and classroom learning. *Assessment in Education*, 1998.

[22] T. A. Angelo and K. P. Cross *Classroom assessment techniques: A handbook for college teachers* (2nd ed.). Jossey-Bass, San Francisco, 1993.

[23] E. Gamma, R. Helm, R. Johnson, and J. Vlissides *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, Boston, 1994.