CEIS420

Diallo Williams

Professor Omar Haddad

Date: 8/22/2023

Final Project- Implementing the Iterator Pattern

Iterator Pattern

Implementing the iterator pattern in software is analogous to having a key that unlocks the huge possibility of traversing collections and aggregations. This pattern provides an organized and efficient method of gaining access to elements within objects without revealing their underlying characteristics. It not only simplifies the code but also increases its flexibility by encapsulating the traversal mechanism. Using the iterator pattern helps developers to divorce their client code from the underlying data structure, making it easier to respond to changes and improving maintainability. This method of iteration is a significant tool in the arsenal of design patterns, providing elegance and organization in dealing with diverse types of collections.

Comparison of OOP, functional, and structured programming languages

Object-oriented programming (OOP), functional programming, and structured programming are three separate programming paradigms, each with its own specific features. OOP emphasizes class and object-oriented programming, emphasizing encapsulation, inheritance, and polymorphism to model real-world entities and their interactions. On the other hand, functional programming is based on using pure functions and immutability, treating computation as the evaluation of mathematical functions, simplifying code reasoning, and allowing for compact, expressive programs. Structured programming is concerned with breaking down programs into smaller, reusable modules, using control structures such as loops and conditionals to construct unambiguous, step-by-step algorithms. While OOP is ideal for modeling large systems, functional programming excels in parallelism and data processing, while structured programming encourages maintainability through a clear program structure. The paradigm of choice.

Design Patterns

Design patterns are recurrent answers to typical software design problems that aid developers in creating code that is maintainable, flexible, and scalable. The Iterator pattern, for example, allows you to retrieve elements of a collection progressively without exposing the underlying structure. It entails creating an Iterator interface with methods such as 'next()' and 'hasNext()' that is implemented by concrete iterators for individual collections. Clients can use these iterators to traverse the elements of the collection without knowing its internal representation, promoting encapsulation, and decoupling between client code and collections. This pattern is very beneficial when working with different data structures since it standardizes the way items are accessed, resulting in more modular and easier-to-maintain code.

Code of report using iterator:

***import java.util.ArrayList;***

***import java.util.Iterator;***

***import java.util.Scanner;***

***class SalesPerson {***

***private String name;***

***private String title;***

***private ArrayList<Double> sales = new ArrayList<>();***

***public SalesPerson(String name, String title) {***

***this.name = name;***

***this.title = title;***

***}***

***public void addSale(double sale) {***

***sales.add(sale);***

***}***

***public double totalSales() {***

***double total = 0;***

***for (Double sale : sales) {***

***total += sale;***

***}***

***return total;***

***}***

***public double minSales() {***

***return sales.isEmpty() ? 0 : Collections.min(sales);***

***}***

***public double maxSales() {***

***return sales.isEmpty() ? 0 : Collections.max(sales);***

***}***

***public double averageSales() {***

***double total = totalSales();***

***return total / sales.size();***

***}***

***public Iterator<Double> iterSales() {***

***return sales.iterator();***

***}***

***public String getName() {***

***return name;***

***}***

***public String getTitle() {***

***return title;***

***}***

***}***

***public class SalesTrackingProgram {***

***public static void main(String[] args) {***

***Scanner scanner = new Scanner(System.in);***

***ArrayList<SalesPerson> salesPeople = new ArrayList<>();***

***for (int i = 0; i < 3; i++) {***

***System.out.print("Please enter sales person name: ");***

***String name = scanner.nextLine();***

***System.out.print("Please enter your sales person title: ");***

***String title = scanner.nextLine();***

***SalesPerson salesPerson = new SalesPerson(name, title);***

***System.out.print("How many sales values will you enter for this sales person? ");***

***int numSales = scanner.nextInt();***

***scanner.nextLine(); // Consume newline character***

***for (int j = 0; j < numSales; j++) {***

***System.out.print("Please enter sales figure for " + name + ": ");***

***double sale = scanner.nextDouble();***

***scanner.nextLine(); // Consume newline character***

***salesPerson.addSale(sale);***

***}***

***salesPeople.add(salesPerson);***

***}***

***double companyTotalSales = 0;***

***for (SalesPerson salesPerson : salesPeople) {***

***System.out.println("\nSales person: " + salesPerson.getName());***

***System.out.println("Total Sales: $" + String.format("%.2f", salesPerson.totalSales()));***

***System.out.println("Min Sales: $" + String.format("%.2f", salesPerson.minSales()));***

***System.out.println("Max Sales: $" + String.format("%.2f", salesPerson.maxSales()));***

***System.out.println("Average Sales: $" + String.format("%.2f", salesPerson.averageSales()));***

***System.out.println("----------------------------------------------------");***

***companyTotalSales += salesPerson.totalSales();***

***}***

***System.out.println("\nCompany total sales: $" + String.format("%.2f", companyTotalSales));***

***}***

***}***A screenshot of a computer

Description automatically generated

***A screenshot of a computer

Description automatically generated***

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

Link to GitHub: https://github.com/DialloWill/My-Final-Project.git

In conclusion, implementing the iterator pattern into software design is a beneficial strategy for improving code organization and maintainability, particularly when dealing with multiple collection kinds. This pattern enhances code flexibility and readability by encapsulating traversal behavior and providing a uniform interface for accessing elements within collections. Furthermore, design patterns, such as the iterator pattern, are important tools in the software development process, helping developers quickly address typical design difficulties. While object-oriented programming, functional programming, and structured programming are distinct paradigms, each has its own set of strengths and areas of applicability, therefore the choice of paradigm is determined by the unique objectives and goals of a software project.