Iterator Pattern

Objective 1:

1. In the given onlineGDB link, the class of pattern is Iterator Design Pattern.

4. The NameIterator class is private scope, and it is declared inside the NameRepositoryType1 class.

8. The "next" method is an "Object" since it is supposed to return the next element in the list, regardless of its type. Since the elements in the list can be of different types, the "next" method is declared to return an "Object", which is the parent class of all Java classes.

Objective 3:

2. I created an ArrayList<String> variable to store the list of names, and initialize it in the constructor by adding the names using the add() method. Then, I modified the NameIterator class to use the size() method of the ArrayList to determine whether there are more elements to iterate over and the get() method to retrieve the current element.

I also added an int variable to keep track of the current index and modified the toString() method to use the get() method to return the current element. I also implemented the Iterable interface and provided an implementation of the iterator() method that returns an instance of the modified NameIterator class.

Objective 4:

3. The Iterator pattern can be useful when you need to iterate over a collection of elements without exposing its internal structure or implementation details. Also, you can use the Iterator pattern when different types of iteration are required for the same collection.

Strategy Pattern

Objective 1:

1. The Strategy pattern is a behavioral design pattern that defines a family of algorithms, encapsulates each one, and makes them interchangeable when at runtime.

4. This particular setup breaks the open-closed principle because the ShoppingCart class is not open for extension, but rather closed for modification. The checkOut method of the ShoppingCart class uses a switch statement to determine the payment method, which means that adding a new payment method would require modifying the ShoppingCart class. This violates the principle since it maintains that classes should be open for extension but closed for modification. You could move the pay methods into separate classes to abide to the open-closed principle.

Objective 3:

3.

- A game that allows players to select different difficulty levels which would then change the game's behavior, for example spawning more enemies with more health
- A web application that allows users to choose different file compression algorithms when uploading files

Adapter Pattern

Objective 1:

1. The Adapter pattern is a structural design pattern that allows incompatible interfaces to work together.

7. They have different interfaces or methods. The GeometricShape interface has methods area(), perimeter() and drawShape(), while the Shape interface has methods draw(), resize() and description().

Objective 3:

2. The Adapter pattern was hardest for me to understand because it requires adapting one interface to another interface that's entirely not compatible. I'd have to understand both interfaces, as well as creating an adapter class that can translate between the two. The adapter class must also be able to handle any differences in functionality or behavior between the two interfaces. Meanwhile, the Iterator pattern was easiest for me since it just involves iterating through a collection of objects using a common interface. It provides a standardized way to iterate through a collection of objects and makes adding new types of collections without changing the code that uses them easier overall.
3. Adapter pattern would be useful when integrating two systems that were not originally designed to work together. For example, let's say eBay uses an older legacy system to store and manage customer data, but they want to integrate this data with a newer system they obtained from Amazon, which requires a different format or interface. Instead of completely overhauling the older system, eBay could use an Adapter to convert the data from the older format to the newer format in a way that the newer system can understand. This would allow the two systems to seamlessly communicate and work together without having to modify the existing codebase of the older system.

Group members: Eric Zhao, James Bebarski