

Task A

Student	First Name	Last Name	ID
1	Yong Loong	Ang	yonan994
2	Jarrett Shan Wei	Yeo	jarye821

1. You should describe which elements implement the Strategy design pattern, and relate the concepts in that design pattern to the example code given. You should be able to reason with respect to both the basic example on GitLab and the elaborate one given above.

Component	GitLab Example	Given Example
Abstract Strategy Interface	Enumerable enumerable	IEnumerable
Actual Implementations / Concrete Strategies	Enumerable evenNumbers/oddNumbers	IEnumerable evenNumbers

Both implement the Strategy pattern because the actual algorithms of even or odd numbers is a family of algorithms which are interchangeable and thus there is a need for different variations. The classes thus only differ only in their behaviour.

2. You should explain how the current implementation violates the principles of the Iterator design pattern (applicable to both the basic example on GitLab and the elaborate one given above).

In the Iterator design pattern, we need to have an Aggregate interface, Concrete Aggregate classes, Iterator interface and Concrete Iterator classes. However, in both examples, the ConcreteIterators do not exist and the Concrete Aggregate uses a hard-coded operation of determining even/odd numbers instead. Since the Iterator pattern should not expose its underlying representation/structure, the pattern is thus violated since we know how the even/odd number is coded. We need to abstract the Iterator so that we can just use `hasNext()` and `next()` methods without knowing the actual code.

3. You should correct the implementation provided so that it works as intended. Explain why it did not work initially.

The first issue is that the program starts by creating a general iterator, then tries to use that iterator to create one that runs through even numbers and another that runs through odd numbers. However, the `where(IPredicate)` function did not return a new instance of the enumerator. It instead adds the predicate filter to the old enumerator and passes it back using the same reference. This results in both `evenNumbers` and `oddNumbers` pointing to the same object which they had both applied their predicate filters. Thus, whenever `next()` is called, the code will skip all the even numbers and odd numbers.

This is fixed in the `where()` function where we create a new temporary `Enumerable<T>` object with local variable name `newEnumerable` and returning that new instance of `newEnumerable` for each `evenNumbers` and `oddNumbers`. This will allow both predicate filters to not have any conflict with each other.

The second issue is the Even number 7 output. In some instances, the program would traverse the entire `ArrayList` and return the last element. However, the element is not checked if the predicate filter matches it before returning it.

This was fixed by ensuring the element passed the predicate filter. A check at the end of the `next()` function is to be added.

4. You should explain how you encapsulated all objects in interfaces to enforce Interface Segregation, which states that all clients should use client specific interfaces. This assumes that all methods accessible through an interface is useable by any client. In your explanation, provide sufficient code to explain how to extend the current implementation to the elaborate functionality given in the example above, by providing interfaces between the different classes.

In the given examples, `IAction` and `IPredicate` are abstract interfaces which classes can implement their own methods. It is thus up to the client to specify exactly how the methods are implemented.

We used interfaces to allow us to pass interface objects as parameters in the methods of the `Enumerable` class, viz. `where(IPredicate<T> predicate)` and `foreach(IAction<T> action)`. We can thus pass in the reference of the interface and polymorphically use the methods of the concrete classes of `predicate.accept()` and `iterator.next()` without knowing the actual implementation of the methods.

Furthermore, these concrete objects implement small interfaces that other classes are able to use to define functionality, thus no client should be made dependent on methods it does not use.

An example to extend the implementation is through a new `ISkipping` interface to enable to skip/step iteration. The concrete class `evenSkipping` implements the interface with its own `skipping()` method. In this case, we can specify the `numOfSkips` which is the steps or number of elements we skip per iteration.

```
public interface ISkipping<T> {  
  
    public boolean skipping(Integer element);  
}  
  
public class evenSkipping implements ISkipping<Integer>{  
  
    private int numOfSkips;  
  
    public evenSkipping(int numOfSkips){  
        this.numOfSkips = numofShips;  
    }  
  
    public boolean skipping(Integer element){  
        // compare and return value  
        return false;  
    }  
}
```