DESIGN PATTERNS HW3

Betülnaz Hayran-28354853660 Github:Naz-bh

Batuhan Kesikbaş-40573251614 Github:BatuhanKesikbas

1-Statement of Work

There are two restaurants that have different menus and they are merging together. The new restaurant uses the first restaurant’s menu for Breakfast and the other’s for the Lunch menu. Unfortunately, the representations of the menus are in different data structures. One uses an array to hold its menu items and the other uses an arrayList.

In order for the waiter at the new merged restaurant to use both menus he will have to translate each menu into something he can use, but in order to translate them different processes have to be used for each menu (because of the different data structures).

This means that there is a violation of encapsulation (the waiter needs to know each menu’s details), that there are no interfaces (coding to concrete implementations) and there is a lot of duplication. In addition to this, if a change is made, there will be a lot of necessary changes to existing code.

To avoid this problem we have used Composer and Iterator Design Patterns together.

2- Explanation on Utilized Design Patterns

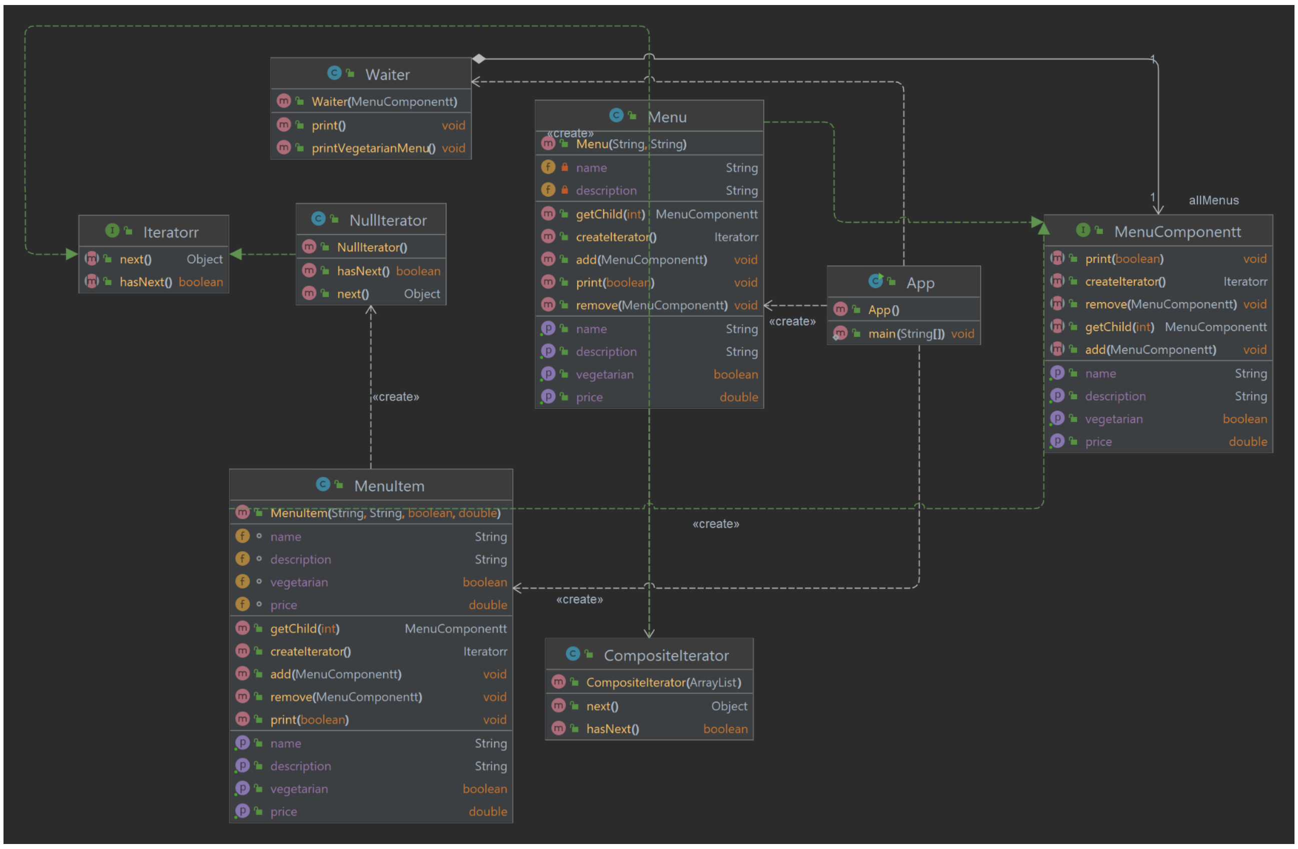
The Iterator Pattern: There are two restaurants that have different menus and they are merging together. The new restaurant uses the first restaurant's menu for Breakfast and the other's for the Lunch menu. There is an iterator interface which means that the classes that hold the data structures have to be able to tell the class that uses the data:

- If they have a next item
- What that next item is (if they have one)

The Composite Pattern: This pattern allows you to treat a tree-like structure of a group of objects (with sub-components or a composition) in the same way. So it allows an iterator to traverse the tree structure (it visits each node and then any children it might have before moving on to the next node on the same level, depth first). This allows us to use different foods on our menus.

The Composite Design Pattern was a pattern that made sense to do alongside the Iterator pattern, that's because it's the most useful when used alongside the Iterator pattern

3- UML Class Diagram



4- Research

The Iterator Pattern

```
public interface Iterator {
  public boolean hasNext();
  public Object next();
}

public interface Container {
  public Iterator getIterator();
}

public class NameRepository implements Container {
  public String names[] = {"Robert" , "John" ,"Julie" , "Lora"};

  @Override
  public Iterator getIterator() {
    return new NameIterator();
  }

  private class NameIterator implements Iterator {

    int index;

    @Override
    public boolean hasNext() {

      if(index < names.length){
        return true;
      }
      return false;
    }

    @Override
    public Object next() {

      if(this.hasNext()){
        return names[index++];
      }
      return null;
    }
  }
}

public class IteratorPatternDemo {

  public static void main(String[] args) {
    NameRepository namesRepository = new NameRepository();

    for(Iterator iter = namesRepository.getIterator(); iter.hasNext();){
      String name = (String)iter.next();
      System.out.println("Name : " + name);
    }
  }
}
```

Usage of the Iterator Pattern:

We created an Iterator interface which narrates the navigation method and a Container interface which returns the iterator . Concrete classes implementing the Container interface is responsible to implement Iterator interface and use it

The IteratorPatternDemo class uses NamesRepository, a concrete class implementation to print Names stored as a collection in NamesRepository.

Step 1- Create interfaces.

Step 2- Create a concrete class implementing the Container interface. This class has an inner class NameIterator implementing the Iterator interface.

Step 3- Use the NameRepository to get an iterator and print names.

Source: https://www.tutorialspoint.com/design_pattern/iterator_pattern.htm