|  |  |
| --- | --- |
| Course | Advanced Software Design – CS525 |
| Assignment | Lab 4 |
| Week | 04 |
| Due | 7/17/2025 |
| Student name | Toe Toe Aung |
| Student ID | 618090 |

1. An AgeIterator that iterates over the customer collection and returns the customer with the lowest age first, then the customer with the 2nd lowest age

**public** **class** AgeIterator **implements** Iterator<Customer> {

**private** List<Customer> list;

**private** Iterator<Customer> it;

**public** AgeIterator(List<Customer> list) {

**this**.list = list;

Collections.*sort*(list, Comparator.*comparingInt*(Customer::getAge));

**this**.it = **this**.list.iterator();

}

@Override

**public** **boolean** hasNext() {

**if** (it.hasNext())

**return** **true**;

**return** **false**;

}

@Override

**public** Customer next() {

**return** it.next();

}

}

b) A filter iterator that can filter out customers based on their address. Using this filter

iterator show the following customers:

1. All customers from Chicago

2. All customers whose zip code starts with “12”

**public** **class** FilterIterator **implements** Iterator<Customer> {

**private** List<Customer> list;

**private** Iterator<Customer> it;

**public** FilterIterator(List<Customer> list) {

**this**.list = list;

**this**.it = **this**.list.iterator();

}

@Override

**public** **boolean** hasNext() {

**if** (it.hasNext())

**return** **true**;

**return** **false**;

}

@Override

**public** Customer next() {

**return** it.next();

}

**public** **void** filterByCity(String city) {

**this**.list.stream().filter(e -> city.equals(e.getAddress().getCity())).collect(Collectors.*toList*()).forEach(System.***out***::println);

}

**public** **void** filterByZipcode(String zip) {

**this**.list.stream().filter(e -> e.getAddress().getZip().startsWith(zip)).collect(Collectors.*toList*()).forEach(System.***out***::println);

}

}

c) An iterator that iterates only over the customers, but after every customer it skips the

next customer.

**public** **class** SkipIterator **implements** Iterator<Customer> {

**private** List<Customer> list;

**private** Iterator<Customer> it;

**public** SkipIterator(List<Customer> list) {

**this**.list = list;

**this**.it = **this**.list.iterator();

}

@Override

**public** **boolean** hasNext() {

**if** (it.hasNext())

**return** **true**;

**return** **false**;

}

@Override

**public** Customer next() {

**return** it.next();

}

**public** **void** skipIterate() {

**while**(it.hasNext()) {

System.***out***.println(it.next());

**if** (it.hasNext())

it.next();

}

}

}

**d) The class diagram without using the composite pattern**

A black and white diagram

AI-generated content may be incorrect.

**e) Class diagram with composite pattern**

A diagram of a computer

AI-generated content may be incorrect.

f)

**Advantages**

* Clients can treat individual objects and compositions of objects uniformly using a common interface
* The request is forwarded to child components

**Disadvantages**

* Leaf classes may be forced to implement methods
* Due to all components following the same interface, the pattern can lead to designs where it's difficult to enforce certain constraints

g) **Class diagram of WebShop application**

**A diagram of a function

AI-generated content may be incorrect.**

public class Product implements ProductCatalogComponent{

private String name;

protected double price;

public Product(String name) {

this.name = name;

}

public double getPrice() {

return price;

}

public void setPrice(double price) {

this.price = price;

}

@Override

public void print() {

System.*out*.println("Product: " + this.name + ", price: " + getPrice());

}

}

**public** **class** Category **implements** ProductCatalogComponent {

**private** String name;

**private** List<ProductCatalogComponent> children = **new** ArrayList<>();

**public** Category(String name) {

**this**.name = name;

}

**public** **void** addProductComponent(ProductCatalogComponent product) {

children.add(product);

}

@Override

**public** **void** print() {

System.***out***.println("Category: " + name);

**for** (ProductCatalogComponent child : children) {

child.print();

}

}

}

**public** **interface** ProductCatalogComponent {

**public** **abstract** **void** print();

}

**public** **class** WebShopApplication {

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

Category shop = **new** Category("Shop");

Category book = **new** Category("Book");

Category grocery = **new** Category("Grocery");

Category milk = **new** Category("Milk Products");

Category vege = **new** Category("Vegetables Products");

Product lord = **new** Product("Lord of the Rings");

lord.setPrice(49.99);

Product cream = **new** Product("Whipping cream");

cream.setPrice(4.59);

Product cabbage = **new** Product("Fresh Green Cabbage");

cabbage.setPrice(3.32);

shop.addProductComponent(book);

shop.addProductComponent(grocery);

book.addProductComponent(lord);

grocery.addProductComponent(milk);

grocery.addProductComponent(vege);

milk.addProductComponent(cream);

vege.addProductComponent(cabbage);

shop.print();

}

}

**ConsoleLogs**

Category: Shop

Category: Book

Product: Lord of the Rings, price: 49.99

Category: Grocery

Category: Milk Products

Product: Whipping cream, price: 4.59

Category: Vegetables Products

Product: Fresh Green Cabbage, price: 3.32