HACETTEPE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
BBM104 PROGRAMMING LABORATORY
2017/2018 SPRING
EXPERIMENT 3
Subject: Inheritance, Polymorphism OOP, Java
Submission Date: 11.04.2018
Due Date: 25.04.2018
Advisors: Dr. Gönenç Ercan, Dr. Öner Barut, Dr. Cumhur Yiğit Özcan, Dr. Ali Seydi Keçeli,
R.A. Pelin Canbay

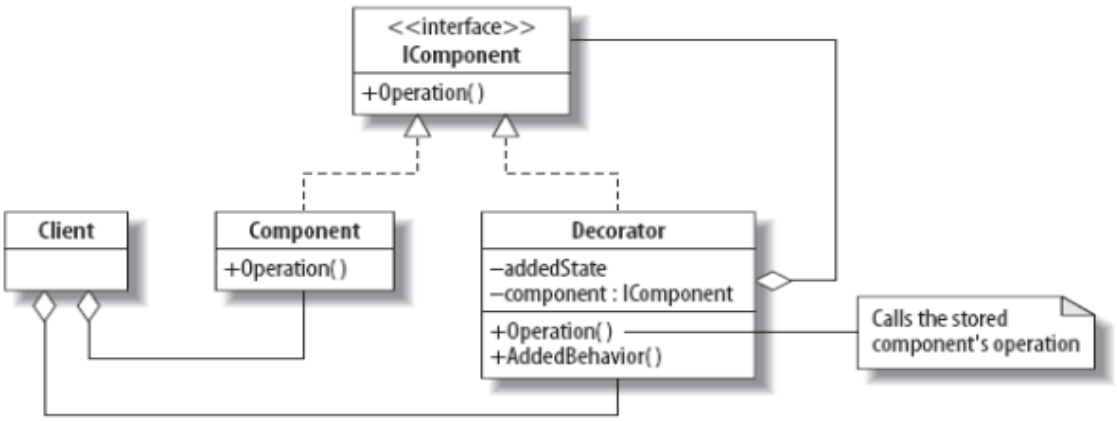
INTRODUCTION / AIM

The aim of this experiment is to introduce you object-oriented programming and design with Java. You will learn the structure of a class, how classes interact, inheritance and polymorphism and basic input-output operations in Java. There will be two main parts to this assignment. One part consists of creating the system's input-output interface and implementing the logic for this interaction. The other part is defining (and implementing) the persistence backend (i.e. how we store and retrieve the information our system uses). You should also use decorator design pattern in first part of the experiment. Your homework will take commands from an input file then it will print the results of these commands to an output file. You should use Data Access Objects (DAO) to manage your data.

Decorator Pattern:

The Java language provides the keyword extends for subclassing a class. Those with enough knowledge of object-oriented programming know how powerful subclassing, or extending a class. By extending a class, you can change its behavior. The Decorator Pattern is used for adding additional functionality to a particular object as opposed to a class of objects. With the Decorator Pattern, you can add functionality to a single object and leave others like it unmodified. Any calls that the decorator gets, it relays to the object that it contains, and adds its own functionality along the way, either before or after the call. This gives you a lot of flexibility since you can change what the decorator does at runtime, as opposed to having the change be static and determined at compile time by subclassing. Since a Decorator complies with the interface that the object that it contains, the Decorator is indistinguishable from the object that it contains. That is, a Decorator is a concrete instance of the abstract class, and thus is indistinguishable from any other concrete instance, including other decorators. This can be used to great advantage, as you can recursively nest decorators without any other objects being able to tell the difference, allowing a near infinite amount of customization.

Decorators add the ability to dynamically alter the behavior of an object because a decorator can be added or removed from an object without the client realizing that anything changed. It is a good idea to use a Decorator in a situation where you want to change the behavior of an object repeatedly (by adding and subtracting functionality) during runtime.



The dynamic behavior modification capability also means that decorators are useful for adapting objects to new situations without re-writing the original object's code.

The code for a decorator would something like this:

```
void operation {
        // any pre-processing code goes here
                component.operation() // delegate to the decor
        // any post-processing code goes here
}
```

Note that the decorator can opt to not delegate to the decor, if, for instance, some condition was not met.

Data Access Object

The Data Access Object (DAO) layer is an essential part of good application architecture. Business applications almost always need access to data from databases or data files. The Data Access Object design pattern provides a technique for separating object persistence (stored data) and data access logic from any particular persistence mechanism (writing to files or accessing a real database). There are clear benefits to this approach from an architectural perspective. The Data Access Object approach provides flexibility to change an application's persistence mechanism over time without the need to re-engineer application logic that interacts with the Data Access Object tier. The Data Access Object design pattern also provides a simple, consistent API for data access that does not require knowledge of sub mechanism. A typical Data Access Object interface is shown below. (This is an example, your interface should look like different)

```
public interface CustomerDAO {
        public void add(Customer customer);

        public void update(int id, Customer customer);

        public void delete(int id);

        public Customer[] getAll();

        ……..
}
```

Experiment:
In this experiment, you are expected to develop a simple Pizza *Restaurant System.* Your application should support the following features:

New Customer:
Create a new customer. Attributes of a customer are:

- Customer id (Unique)
- Customer Surname
- Customer Name
- Customer Address
- Customer Phone Number

Remove Customer:
Remove a customer from the database by using its identification number.

List Customers:
Print the list of customers ordered by name.

Create Order:
Create a new order for an existing customer and append it to order data file. Each order has a unique order id.

Add Pizza:
Add a pizza to a given order with given toppings. A pizza can include 3 toppings at most. There are 2 types of pizzas. These are:

- American Pan Pizza 5 $
- Neapolitan Pizza 10 $

A pizza can include 3 toppings at most. There are 4 types of toppings.

- Soudjouk 3 $
- Salami 3 $
- Pepper 1 $
- Onion 2 $

Add Drink:
Add a soft drink to an order (2$). There is the only cola in this restaurant.

Pay Check
Calculate the paycheck of an order. List all pizzas and drinks in the order then calculate the total cost of an order.

Customer information and order information will be stored in text files (lists or arrays will also be accepted). The format of the files for persistence are shown below. Names of these files will be taken as arguments in your application. You can reach sample customer file, order data file and sample input-output files from the page of piazza site of the experiment.

Customer Data File Format

<customerID> <customerName> <customerSurname> <phone number> Address: <address>

Customer name, surname, and phone number consist of one string. The address can be made up of one or more strings. Entries in customer data file should be ordered by ID.

Order Data File Format

Order: <orderID> <customerID>
<pizza type> (Neapolitan or American Pan) <topping [1-3]> (There can be 3 toppings at most)
<drink>

Entries in order data file should be ordered by ID. After add and remove operations, all changes should be reflected data files.

Input File Format (Note: Number of topping can vary from 1 to 3.)

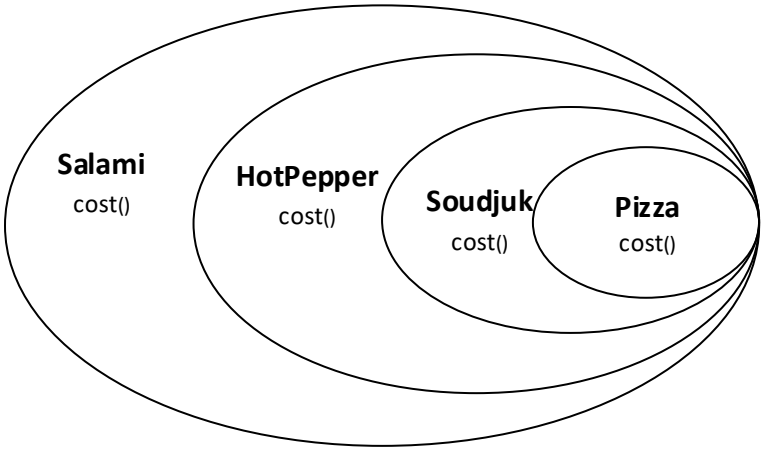
- AddCustomer <cutomerID> <name> <surname> <phone number> <address>
- RemoveCustomer <cutomer ID>
- CreateOrder <OrderID> <CustomerID>
- RemoveOrder <OrderID>
- AddPizza <OrderID> <pizza type> <topping>
- AddDrink <OrderID>
- PayCheck <OrderID>
- ListCustomers

Output File Format

- Customer <CustomerID > <name> added
- Customer <CustomerID> <name> removed
- Order <OrderID> created
- <pizza type> pizza added to order <OrderID>
- Drink added to order <OrderID>
- PayCheck for order <OrderID>
  - ✓ <pizza type> <topping[1-4]> <cost> $
  - ✓ <pizza type> <topping[1-4]> <cost > $
  - ✓ SoftDrink <cost> $
  - ✓ Total: <total_cost> $

Implementation Details:

You should use decorator pattern **(20points)** for add topping mechanism. Add topping code should look like this:



pizza.addTopping(new Salami(new HotPepper(new Soudjuk()));
This a pizza with salami, hot pepper, and soudjouk.

pizza.printToppings(); Output of this method call should be ***Soudjuk HotPepper Salami***

pizza.cost(); Output of this method call should return the costs of pizza and its toppings

The advantage of using data access objects is the relatively simple and rigorous separation between two important parts of an application which can and should know almost nothing of each other, and which can be expected to evolve frequently and independently. Changing business logic can rely on the same DAO interface, while changes to persistence logic do not affect DAO clients as long as the interface remains correctly implemented. In the specific context of the Java programming language, Data Access Objects as a design concept can be implemented in a number of ways. This can range from a fairly simple interface that separates the data access parts from the application logic to frameworks and commercial products. DAO coding paradigms can require some skill.
For our system, we will store the data inside the individual files (one for order and one for customer class), and use Data Access Objects (DAO) to manage them **(10points)**. Each DAO will have the basic operations to manage the data (defined by the DAO interface provided with the assignment). So basically what you need to do is implementing following abstract methods:

***Object* getByID(int ID)** // read a single entry from the file
***Object* deleteByID(int ID)** // delete a single entry from file
***void* add(Object object)** // add or update an entry
***void* update(Object object)** // add or update an entry
***ArrayList* getALL()** // get all entries
You could add additional methods to your DAOs and you can also make changes in DAO interface. Your application should terminate and save the files automatically when finish to reading input file.

SUBMISSION

Your submission will be in the format below.

```
< StudentID>
        |-- report
                report.pdf
        |-- source
                Main.java
                *.java
```

Notes and Restrictions

- Regardless of the length, use UNDERSTANDABLE names to your variables, classes and Functions. The names of classes, attributes and methods should obey Java naming convention.
- **Execution of application will be 40 points.**
- Write READABLE SOURCE CODE block and comments (Readability and modularity) **(10 points)**
- All file names are fixed. **(You can use Lists or Arrays instead of files)**
- Save all your work until the assignment is graded.
- Report will be 20 points
- You should give your detailed design in solution (with UML class diagram) **(10points)** in the report.pdf. In solution part of your report explain structures of your classes and how they attract each other.
- Please explain your design, data structures and algorithms **(10points).** Give necessary details without unnecessary clutter.
- Guide: ftp://ftp.cs.hacettepe.edu.tr/pub/dersler/genel/FormatForLabReports.doc
- All assignments must be original, individual work. Duplicate or very similar assignments are both going to be considered as cheating.
- You have to use "Online Experiment Submission System". http://submit.cs.hacettepe.edu.tr. Another type of submissions WILL NOT BE ACCEPTED.
- Submission deadline is 25.04.2018, 23:55.
- Don't use packages.

REFERENCES

- https://www.tutorialspoint.com/design_pattern/decorator_pattern.htm
- https://www.tutorialspoint.com/design_pattern/data_access_object_pattern.htm
- http://java.sun.com
- http://java.sun.com/j2se/1.5.0/docs/api/java/util/Scanner.html
- http://tasarimdesenleri.wordpress.com/2009/09/15/
- http://en.wikipedia.org/wiki/Class_diagram
- http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell