****

**OBJECT ORIENTED PRINCIPLES**

ASSIGNMENT 3: A JAVA APPLICATION FOR MANAGING A STORE.

**Declaration of Authorship**

I, Rinalds Matvejevs, declare that the work presented in this assignment titled ‘A Java Application for Managing a Store’ is my own. I confirm that:

* This work was done wholly by me as part of my BSc. (Hons) in Software Development, my Msc at Munster Technological University.
* Where I have consulted the published work and source code of others, this is always clearly attributed.
* Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this assignment source code and report is entirely my own work.

On 7/12/2024

Signature: R.M

**Java Application Description.**

The Hot Chocolate Truck is a Java application that provides a long menu containing some of the functionality you would expect from a hot chocolate truck. Some of these functionalities includes:

* A system for adding and removing customers, drinks, and orders both to and from the truck.
* Switch cases to view drink and customer information.
* Giving and removing customers money for ordering drinks.
* Adding and removing drinks to the truck's menu.
* Managing the creation of orders, giving a drink to an order and giving an order to a customer.
* Displaying details of the order.
* Saving and loading data for customers, drinks, and orders using serialization for each.

**Technical Difficulty: OOP Concepts Demonstrated in the Java Application.**

1. **Primitive and Reference Variables:**

* The Drinks class uses double price as a primitive variable to store the price of a drink, while String name is a reference variable for the drink's name.

1. **Classes and Objects:**

* The class Customer models a customer of the hot chocolate truck. The method HotChocolateTruck::addCustomer creates a new Customer object.

1. **Encapsulation:**

* The Customer class has private fields, such as balance, and public methods, like getBalance and setBalance, to access and update the field from other classes.

1. **Aggregation:**

* The Order class has a private field drinks which is an ArrayList<Drinks>, representing a collection of drinks in an order.

1. **Inheritance:**

* The classes CadburyChocolate, and KitkatChocolate inherit from the abstract Drinks class, allowing them to have different unique qualities while both being drinks.

1. **Class Hierarchy:**

* CadburyChocolate, KitkatChocolate, also inherit from the abstract Drinks class, forming a class hierarchy for different types of hot chocolate.

1. **Static Polymorphism (Overloading):**
2. **Dynamic Polymorphism (Overriding):**

* The Drinks class and its subclasses override the toString method, ensuring that the output reflects specific information about each drink type.

1. **Abstract Class:**

* The Drinks class is abstract, containing the abstract method getDescription(), which must be implemented by each subclass like CadburyChocolate or KitkatChocolate.

1. **Interface:**

* The HCTInterface class is what models the truck's operations, and it is implemented by the HCTImplement (Hot Chocolate Truck Implementation) class, which completes the functions.

1. **User and Developer Isolation:**

* The interaction with the system is abstracted by the HCT interface, and the HTC implementation class hides the internal details of managing customers and orders.

1. **Upcasting:**

* In the HotChocolateTruckImp::addDrink method, drink objects are treated as their parent type (Drinks) when they are added to an order, showing use of upcasting.

1. **Static Fields and Methods:**

* The Order class uses a static field, orderIDCounter to generate unique IDs for every order, making sure there are no duplicates.

1. **Final Fields, Methods, and Classes:**

* The Drinks class has final fields like name, ensuring that it cannot be modified after initialization.

1. **Data Structures:**
2. **Java Generics:**

* The use of generics allows the ArrayList<Customer> and ArrayList<Drinks> to maintain a list of specific types of objects, ensuring type safety.

1. **Downcasting:**

* When accessing a drink in the Order class, it is downcasted from Drinks to specific subclasses like CadburyChocolate, allowing access to subclass-specific methods.

1. **Exception Handling:**

* The application employs try-catch code to validate user input in the menu, preventing invalid data from crashing the system and enhancing user experience.

1. **File Reading and Writing:**

* The HotChocolateTruckImp::saveDataToDisk and HotChocolateTruckImp::loadDataFromDisk methods use serialization to save and load customer and order data from text files.

1. **Default Constructor and Copy Constructor:**

**UML Design: Java Application.**

A diagram of a company

Description automatically generated

Legend:

A diagram of a truck

Description automatically generated

**A diagram of a computer

Description automatically generated**

**Programmer 1: User**

**A diagram of a company

Description automatically generated**

**Programmer 2: Developer**

**Testing the Java Application.**

The application was tested interactively in the MyMain class by performing several sessions, testing out a different function each time. Users can select from various commands to:

* Add customers to the truck.
* View customer details.
* Add or remove drinks from the menu.
* Create orders and add drinks.
* Display order details.

Testing the Interface brought good results as the code was both compiling and performing as expected. The data was being saved and passed into the next session with ease when requested. There is no functionality past creating an order such as payment process and completing the order. The drinks inherit and implement from their parent class without causing any problems which was satisfactory.

The menu prints out the name of the customer along with the ID as it made for easier readability without requiring any additional external get requests to remember which user had what id. The order prints out the drink information to see what is within the order, which again helped in organising and remembering the different instances.

Ensuring the implementation of as many OOP concepts as possible was difficult at times as despite planning out the assignment before starting, I found myself having to double back and implement code within classes or functions to incorporate the concepts.

I am glad I kept my example simple as it allowed me to focus on the smaller scale to perfect what I have as a form of quality over quantity. For example, having more drinks or different types of orders such as delivery or pickup would overcomplicate the system.

YOUTUBE LINK!!!!!

https://youtu.be/knASS8XsTO4