**Local restaurant order and billing simulator app**

All the app works via the console with only POJO

The purpose of this app or case of study is to implement an electronic order service for a restaurant, the features of the app are as follows, by using an app like this we are able to have a better control of the orders of the day, the bills and basically al the ordering process:

**Requirements:**

* Create an Order that can hold different type of meals
* Meals can have 3 different combinations, simulating a combo meal request, a simple only food request and a simple only beverage request
* Foods can be created dynamically trough the client
* Beverages are fixed
* The Order must implement an add meal feature, remove meal feature, undo changes feature, and Empty feature.
* From an Order we can create a bill
* A bill consists of individual instances of bills created in regard to the meals contained in an order
* Bills and Orders can be “printed”, this was simulated by creating txt files out of the Orders or Bills
* The app shall remain and be prepared for changes, we can add new foods, meal types, billing types, added functionalities with easiness.
* Combo Meals should have a discount
* Orders with more than 3 meals should have a discount

**Patterns used:**

**Builder Pattern**: this pattern was used for the creation of all foods

**Memento Pattern**: this pattern was used to implement the undo chances functionality to the Order class

**Bridge Pattern:** this pattern was used to create all the interface relations through Order class and meal, Beverage, and Food abstract classes, there was no concrete implementation of this pattern but the relationship structure between meal, Beverage and Food abstract classes is a bridge pattern

**Visitor Pattern:** used to add additional functionalities to the already done classes without altering much of the code, this pattern was used 2 times in the project for different classes. One for the implementation of the “printing” feature, this functionality was added to the Order class and Bill class. and one to add the functionality of price calculation on top of Beverage and Food abstract classes

**Simple factory method:** this method was implemented for the creation of the Bill, Bill consists of specific type of bills that change depending of the meal prepared. A factory was created to create these meals in regard to the List<Meals> from the Order class

**Overview**

Implementations Order

* Most of the outputs used in the code are strings, therefore the main output tool used for storing the states of the order is the toString method, all the order is stored in a string and return in the toString method of Order, by overriding this method in different classes we can call the whole food, meal, order or bill without calling individual getters. Working with this was useful for the “printing” implementation.
* The order class basically is a List for Meals with added functionalities, this class has an add meal method, remove meal, undo method, and undo all (empty meal) method.
* There is a Meal superclass and we have 3 subclasses for each type of combo Meal, in the instance that we need to add new types of meals
* Actual meals for a Combo can be created as Combo(food and beverage), OnlyFood ( a sole meal) , OnlyBeverage (a sole beverage) . If there’s a requirement that needs to add another type of combination we can do it trough here by just simply adding a new subclass.
* There is a Food abstract class used to model all the foods, specific functionalities outside of the food creating are added in here as concrete methods.
* There is a Beverage abstract class used to model all the drinks, specific functionalities outside of the food creating are added in here as concrete methods.
* Beverages subclasses are immutable.
* For now, we have a coke and water class that inherits from the beverage abstract class and we have a burger abstract class inheriting from the Food abstract, which is used to model the concrete subclasses SimpleBurger and CustomBurger. We also have a concrete HotDog class that inherits directly from the food abstract class
* There is also a getStatus method implemented to print directly the actual state of the order
* Order class also implements the accept method which is needed due to the Visitant interface that is being implemented on top of it. This method is used for the txt document creation.

Implementations Bill

* Following the same statelines as Order, the output of bill works with strings, same as Order, a string is generated and all the string outputs of each of the generated bills are stored in it, then this is used with a format in the toString method of Bill
* Bill receives an order and creates de bill from it, the list of meals is extracted and processed in the BillFactory, and then Bill Factory returns a List of Bills. Each bill is generated depending in the type of meal that was iterated through the list of meals of the order. At this moment we have 3 types of Bills, bills for combos, bills for sole burgers and sole beverages, these bills have a “Bills” interface superclass which requests to implement a calculatePrice method and requests to override the toString method.
* The calculatePrice method calls the getActualPrice method of the beverages and Food instances inside a meal. For the combo meal case this is needed to implement a discount to the combo but for the other bill subclasses is trivial however in the case that we need or desire to implement a discount this method might come handy.
* Factory returns the list of Bills and Bills iterates through it in the toString method, and we use a conditional to check if the number of bills is higher than 3 to apply a discount to the whole order
* There is also a getBill method to call directly the toString method and print it in the console.
* Bill class also implements the accept method which is needed due to the Visitant interface that is being implemented on top of it. This method is used for the txt document creation.

**Food and Beverages abstract classes**

* These classes have 1 method for modeling the child classes which is the toString method, and implement the getActualPrice method from the Visitant class implemented due to the visitor pattern. We have 2 getActualPrice methods because one uses the fixed one created for the project and another was created to pass as an argument future types of prices configurations

**Visitor Pattern : getActualPrices**

* This pattern is used to implement a volatile type of pricing for the items which is not hardcoded unto the actual items, the hard coded values are coded inside the PriceGeneratorVisitor class which implements a Visitor interface requesting to implement a generatePrice method for a Food and for a Beverage by using overloading. The generatePrice method checks at runtime if it’s a Food or Beverage and executes accordingly, and inside each overloaded generatePrice method we have the hardcoded values that check if the passed object is of an instance of a specific type of item. In the moment when we want to edit the price of an existing item or add a new price or price configuration for an item we edit it through here.
* The Visitant Interface models 2 methods called getActualPrice which will be the ones associated to the genertePrice method and the Visitant will be implemented to the Beverages and Food Interfaces to be able calculate the Prices for each item, the object will be passed by a “this” to the argument of generatePrice inside the getActualPrices method and a Visitant (PriceGenerator) will be passed to the getActualPrices for an specific priceConfiguration or if we use getActualPrices() a new PriceGeneratorVisitor will be created

This getActualPrices method then gets called at a bills type of bill inside the calculatePrice method

meal.getBeverage().getActualPrice() or

meal.getFood().getActualPrice().

**Visitor Pattern: txtTicketGenerator class / accept**

This pattern is used to implement also a volatile type of ticket generator, but for simplicity only a txt generator was created, it works in the same was as the previous Visitor pattern, visitor models 2 generate methods, one for Order and one for Bill and its implemented unto a txtTicketGeneratorClass.

The txtTicketGenerator class works with the IO libraries and with bufferedWriter and FileWriter classes and which we retrieve the toString of an Order or Bill and use it as an argument to the write method to generate the file.

This class has 2 constructors which will dictate if the class will use a default path or an specific path.

A Visitant class was created to model the method accept for the Order and Bill class and in it we pass a Visitor (txtTicketGenerator) to generate the bill.

When accept gets executed from an Order or Bill, the file will be generated.

***Class and functionalities***

**Client**

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

In the client we create an Order without passing an argument and we also create a Visitor of type txtTicketGenerator after this we create 4 Meals, 2 of type FoodAndBev, 1 of type Only Food and 1 of type Only Beverage and then we add them to order1, then we remove the last meal of order1, we undo this and print the whole orderStatus. After this we use order1.accept(txt) to create a txt file with the visitor created in the path that was passed to the constructor

Then is Bill is created and the order1 is passed to its constructor, after this we print the whole Bill in the console and we create a txt file using the same method as for the order1

**Beverage and Food abstract classes**

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

This abstract classes are practically the same but they were created separately to have the parent class reference.

In here we are implementing the visitant Interface which is needed to implement a visitor pattern created to implement a volatile type of price calculation. Visitant for the beverage and Food contains 2 abstract methods called getActualPrice that will get overloaded depending if a Visitor was passed as an argument or none was passed, if no Visitor was passed the method will create a “default” Visitor of PriceGeneratorVisitor. After creating the visitor or receiving it we will execute its method generatePrice( -------) through the return statement and to this method we will pass “this” indicating that the generatePrice was generated based of the instance of the item created. For example, if we are creating a Coke object, this will return the Coke object which its also a Beverage because we are extending the abstract class to the Coke. generatePrice works at this moment only for Beverages and Foods but can be easily edited through the Visitor class used for this method.

Texto

Descripción generada automáticamente

As mentioned, this is the Visitor interface, which is denoting that 2 methods must be implemented to be used with overloading

The PriceGeneratorVisitor class that implements Visitor is the one below, this is a concrete class

And needs to be changed depending on the actual meal.item classes created. For each new class it will need to be added in this class to get its price calculated. If a new type other than beverage or Food gets created we will need to create another method overloading it and create its corresponding conditionals. For example if we create a new meal.item Interface called dessert we will need to add a new method in Visitor and respectively in here with its conditionals for each subclass type dessert.

Texto

Descripción generada automáticamente

this is the generatePrice method for Beverages, in here we pass a Beverage and depending on the subclass passed to the argument the price will be updated

Texto

Descripción generada automáticamente

This is the generatePrice for food, in here we calculate the Price of each type of concrete food

For SimpleBurger we just give a fixed price, CustomBurger we have a price of 80.00 and gets increased depending of the number of patties the CustomBurger has, for Hotdog we calculate the price depending on its size and quantity of toppings.

**Meal.items**

All Food items are created through a builder pattern, beverages are fixed. All food items constructors are private but have a constructor as a nested class to be able to build the food.

The nested builder class are static to be able to accessed without creating an object out of them. Builder pattern is useful because it gives us full flexibility when creating the object.

Texto

Descripción generada automáticamente

Burger abstract class is used as a base for the concrete burger objects

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

We have the bread/bun as final, the patty also as final and the only editable fields are the toppings and the cheese.

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Imagen que contiene Texto

Descripción generada automáticamente

Same as simple burger but a lot more customizable, for this item the getter for patties was needed to calculate the Price of it, in the previously shown Visitor subclass

Texto

Descripción generada automáticamenteCaptura de pantalla de computadora

Descripción generada automáticamente

HotDog is similar to the burger classes, but it inherits directly from the food abstract class Its price is calculated in regards to the size and quantity of toppings it has. Both fields can be setted through its builder.

**Meal. Items (beverage classes)**

This classes are the ones that inherit from beverage, these are fixed at the moment.

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

**Meal abstract class**

Texto

Descripción generada automáticamente

This class is used to model the child classes and implements getters and setters to be able to access the private fields from other classes, we have an editable mealId that uses the mealCounter to update its value based on the number of meal instances created. The resetMealCounte method is called by the Order constructor to reset the meal Counter when a new order is created. The toString retrieves the toStrings from the Food and beverage objects and depending if an object exists or not we create the respective String format for this meal.

Meal subclasses are the same, with the only difference that the ones having a sole meal, instantiate to null the beverage field, the ones with only a beverage instantiate the meal to null and the ones with both values instantiate both with the values passed to the constructor respectively. In here we also call the setMealId method to set the mealId.

The way that this class relates to the beverage and food is by using a Bridge Pattern, we are basically having a bridge of abstract classes that can be instantiated to any kind of subclass and the Meal abstract class is used as the midpoint between all the classes.

Texto

Descripción generada automáticamente

*This is the subclass for the “Combo” Subclass*

**Memento Pattern implementation**

Texto

Descripción generada automáticamenteTexto

Descripción generada automáticamenteThis pattern was implemented with the objective to implement a rollback feature in the order, which is implemented by the method “undo()” inside Order. The memento pattern consists of 3 parts, a Memento class that works by saving a state of the Order List of Meals, a Caretaker class that’s used as the “backup” managing tool for the states and the “originator class” which is the one that calls the methods from from the Caretaker, in this case the originator is Order. Memento has only 1 method and a constructor that asks for a List<Meal>, the method is to retrieve state from the Memento objected created with the List<Meal>, the caretaker class works as a queue, it is a List<Mementos> and technically a List<List<Meal>>, it has 2 methods a push and pop, when we add or remove a Meal in the Order class, a push method gets invoked and to its argument a new Memento is created with the actual List<Meals> state of the Order class, creating a backup of it. When we do an undo in the Order class we call the pop method and retrieve the last state of the List of Mementos in the caretaker object -1 and instantiate the order listOfMeals field to it, by doing this we are reversing the state of the Order. In our case the Caretaker class is initialized inside Order creating a composition relationship between caretaker and the Order. If the list in caretaker reaches 5, the push method will start deleting index 0 to limit the undos and the memory usage for it

**Order class**

Captura de pantalla de computadora

Descripción generada automáticamenteThis class is the main class that conforms all the project

Texto

Descripción generada automáticamente

We have 4 fields in here, 2 of them which are used for the ID and 2 of them that manipulate the Order. Same as Meal, in here we have a static counter and an Id, which are updated each time an order is created, then we have the caretaker object which is a like a nested backup list inside the Order class which is needed for the Memento Pattern.

When we create an order we initialize an empty listofMeals or if we pass a List<Meals> to the constructor the listOfMeals get initialized with that list, after the list initializing call resetMealCounter, we create an orderId and a new caretaker object

Order has various list manipulation methods, and respectively an accept method which needs to be implemented due to the Visitant Interface it will be discussed further in the document.

With getOrderCounter we can get the Order static counter, and we also have the add meal, removeLastMeal, undo, and empty methods.

When calling addMeal as mentioned previously we will add the new meal into the Order, and create a backup of the full meal list in the caretaker object, as a memento, each time a backup state gets created, this will also work for removeLastMeal in the same way as addMeal.

The undo method just as mentioned in the Memento section will retrieve the last backup state -1 from the caretaker, and initialize the listOfMeals with it, its return type will be of type Order and it will return “this” to be able to chain the undos in the calling environment.

The empty method is literally a clear for the list, but with the added functionality to print in console the state of the Order.

We also have a getter for the list, this will be used when creating the Bill.

Finally, we have getOrderStatus, which prints the toString method of the List and the overridden toString which returns a String of the whole Order Status, we need this to be a string for the added functionality that will be mentioned up next.

Last we have the accept Method, which executes generate( ---- ) and asks for a Visitor class, this is another Visitor pattern with a create to txt functionality which is implemented to the Order and Bill classes.

**Visitor Pattern ticketGenerator**

this pattern same as the previous one consists of 1 concrete class, a Visitor Interface and a Visitant interface which is implemented to Order and Bill and encapsulates a generate method inside an accept method to execute it once a Visitor is passed as the methods argument.

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

The txtTicketGenerator implements visitor and its used to create a txt File of the Order or Bill, the generate method same as before is overloaded and selected at runtime, depending on the actual object passed to the method with a “this”. We use the IO libraries and import BufferedWriter and FileWriter, then in the writer variable we execute write and pass the actual bill or order and execute toString to return string to the write, then we use .close() to make the changes appear in the txt File generated via new FileWriter. For the constructor of the class we have a field called path, this field gets passed to the FileWriter constructor and if we invoke this class without passing an argument to the constructor the default path for the file will be used, which is in the root folder of the project. We create a bill or order, use accept pass a path or none and the file will be created with the order or actual bill of the order.

**Bill Class**

Texto

Descripción generada automáticamente

This class will be used as the main bill that will be printed, this class only has 1 constructor which asks for an Order, and 2 fields, 1 for a List<Meals> and one for a List<Bills> that will be used for the final bill string in toString.

The mealL gets the List<Meal> from the Order passed to the constructor and the bills field invokes a BillsFactory and executes a static method called createBill in which the mealL is passed as an argument and then returns a List<Bills> .

Same as the order we have a getBill method that executes and prints the toString, and we have the toString which has some functionalities. In it we create a new StringBuilder, this will be used to append all the Bills from the List<Bills>, also we created a totalSum which will be used to retrieve the each Meal calculated in each Bills from the List<Meal>, a discount variable was created and a flag needed for this discount when creating the last String.

The method iterates through the List<Bills> and appends each toString to the StringBuilder and at the same time for each cycle the calculatePrice method will be executed adding its value to the totalSum for each cycle.

Then we will check, if the List<Bills> had more than 3 elements a discount o .15 will be applied and a flag will be turned to true indicating to add to the string the discount section.

Lastly we have the accept method from the Visitant Interface.

**Factory Simple method**

Texto

Descripción generada automáticamente

The factory Simple method is fairly simple , the static method createBill gets passed a List<Meal> and we create inside of it a List<Bills>. We will then iterate through the “meals” and check if the actual meal from the cycle is an instance of a Meal Subclass, If we add a new type of meal subclass, for example, FoodAndBeverageAndDessert, we will have to add another conditional in here to check for that particular new type of Meal. If theres a true, we will pass the actual meal of the cycle to the constructor of a subclass of Bills in regards to the actual Meal checked in the conditional statement create a new object of it, and add it to the billList. When the loop finishes we will return the full List<bills> to the Bills class.

**Bills Interface and subclasses**

Texto

Descripción generada automáticamente

Bills will request to implement a calculatePrice method and to overrde the toString method in the child classes

Texto

Descripción generada automáticamente

This is the FoodWithBev Bill for “combos” in here we have 2 fields, a Meal meal which is need to store the meal passed to the constructor and an editable comboDiscount.

For this case, the calculatePrice method will retrieve the getActualPrice method implemented as a Visitor pattern on top of the beverage and food class, for this we need to access the beverage or Food from the meal by using getBeverage or getFood and then getActualPrice, this is done for the beverage and food to be able to be added up, once they are added the comboDiscount is applied and we return the price, this value is the one used for the totalSum local variable in the toString method of Bill, the toString method in here basically creates “the bill” for this meal and returns it as a string, this string later is appended in the toString method of Bill.

For only Beverage and only food is the same, but in the calculatePrice we instead only execute 1 getActualPrice and return its double for each type o Bill respectively, no discounts are applied for these bills.

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

**Conclusion on why I implemented each pattern**

**Builder:** helps in object creation, making its creating much more flexible and without the nuisance of having to work through their constructors, by using them we are able to create complex objects with a lot of attributes much simpler to create and a lot more clean in the calling environment and less clogged with constructor instantiation.

**Memento:** was implemented as a rollback feature

**Bridge:** for bridge I wasn’t planning to actually implement it but once I started doing my individual research on what patterns to implement, I found out I used it without being aware of it. The relation between Food, Meal, Beverage, and Food is almost or literally a bridge pattern.

**Factory Simple:** it is used mainly for the purpose of the OOP principle of 1 single responsibility per class, also it helped in making the Bill class a lot cleaner.

**Visitor:** Visitor same as Factory Simple was implemented with the idea in mind to avoid making the code more difficult to read and to maintain the single responsibility principle.

My only concern was with the Order class which is heavily coded and has a lot of method calls, something I wanted to implement was to extract the similar methods and create a separate functions class for the list manipulation