L1: Pricing

Outline of the approach

In this project, I have used OOP and created three abstract classes (Item, Drink and Food) and other classes (HotDrink, ColdDrink, Sandwich…) to create objects of a specific item on the menu. Each class has most of the code and implementation inherited from its parent classes and only has some unique options and requirements inside. For example, to add an option for all the drinks, we go for the general abstract Drink class, and to add an option like the chocolate sauce for hot drinks, we go for HotDrink class. The object diagram file can be found in the same folder of this document.

Design choice

As I observed, each specific kind of drink may have some unique requirements but overall they have a large number of common attributes and methods so I chose to create a hierarchy of classes to leverage all the features of OOP.

- Calculate1():

  I divided the price of the item into two parts: base price + (the price of the options). In this view, we can see that class Item is the object that has only a base price without options and each subclass of the price structure just includes more options to the base price. This means that I viewed drink has a base price of $2 and the drink type is an option: option hot costs $0, option cold costs $0, and option blended costs $1 (This value is stored in the variable PRICE_ADJUSTMENT of each Drink class - default is $0).

  I decided to keep track of some attributes such as NAME, SIZE, MILK_OPTION,… as **class attributes** so that we don’t have to create the same copy of attributes for each object. For each child object, we can create its own version of class attributes if they differ from those in the parent class. For example, class Drink has a class attribute SIZE = {"S": 0.0, "M": 0.5, "L": 1.0, "XL": 1.5} to keep track of all the prices of the size of the cup and class HotDrink which is a child of Drink is only available for size S and M. Therefore, we create a version SIZE of HotDrink as SIZE = {"S": 0.0, "M": 0.5}. This approach also benefits in the case that if we consider HotDrink as having “premium” ingredients so each ml we add, the increase in the price is not

the same as other drinks and we need to create our own version of SIZE for HotDrink, like SIZE = {"S": 0.5, "M": 1, “L”:1.5} (Note that the prices differ from the one in SIZE of the Drink).

- Calculate2():

  To add a milk tea drink type with a base price of $2.25 I create a new class MilkTea inherited from Drink and set BASE_PRICE to 2.25 instead of inheriting the default value of 2 in the Drink class.

  To add XL size and its cost, I add another key-value pair to the SIZE in Drink class so that now all the drinks cold, blended and milk tea can have an option XL size (except HotDrink because it has its own version of SIZE).

  To add milk options, I created a new Dict MILK_OPTION to store the milk options data in the Drink class so that all the drinks can have access to it, updated the constructor with the parameter milkOption and set the default value to None so that the previous function Calculate1 doesn’t break due to different method signature.

- Calculate3():

  To add the chocolate sauce option for hot drinks, I went to the HotDrink class and update the information also created a helper method getChocolatePrice() to calculate the price of the chocolate pumps due to its special requirements.

- Calculate4():

  I created a new class Food inherited from the class Item since it has different attributes and implementations as the Drink class, then created two classes Sandwich and Bagel inherited from the Food class. Each of the Sandwich and Bagel classes has its own set of toppings.

- Calculate5():

  To create a list of items, we can create objects from one of these classes HotDrink, ColdDrink, BlendedDrink, MilkTea, Sandwich, or Bagel. Each has its own function to calculate the item price and display the price breakdown.

Sample output