

Learning Outcomes

Upon completing this assignment, students should be able to

- **Implement** a hierarchy of *classes*.
- **Program** a linked-list of classes.
- **Use** the *decorator* pattern

Introduction

You have been hired by *Ahab's So-So Coffee* to automate their order system. You will have a *main menu* that has three functions:

- New order
- List orders
- Quit

When the cashier selects *New order*, they get the order a drink menu:

- Coffee
- Espresso
- Latte
- Finish order
- Cancel order

When a drink is ordered (*Coffee* for \$1.50, *Espresso* for \$2.75, *Latte* for \$3.25), then it goes to the decorate a drink menu:

- Add flavor
- Finish drink
- Cancel drink

When the cashier adds a flavor, ask for the name of the flavor and decorate the current drink with it. (All flavors cost \$1.25.)

When the drink is finished, it is appended to the linked-list representing the current order and you go back to the order a drink menu. If the drink is canceled, then do *not* add the drink to the order and just return to the order a drink menu.

When the **order** is finished, print out the order with the description of each drink, the total of each drink, and, at the end, the total of the whole order. Add the order to the linked-list of orders in the program.

If the **order** is canceled, do *not* add it to the list of orders and do not print out the order.

When the cashier selects *List orders*, print out the total of each order, one per line, and, at the end, the grand total across the orders.

Method

Notice that there are *three* (3) different menus used in this program. This means you will create three instances of your *Menu* class, one for each of the levels of menu. Create each menu *once* at the beginning of the program.

Read the entire assignment. Sketch out a short interaction with the system where an order is placed for a *Coffee*, another for an *Espresso* with *hazelnut* and a plain *Hot Chocolate*. Draw which menu is displayed, which selection is made, and what the order list and drink lists look like.

Use this same sort of interaction drawing to handle the other possible menu selections when you get to implementing them.

Getting Started

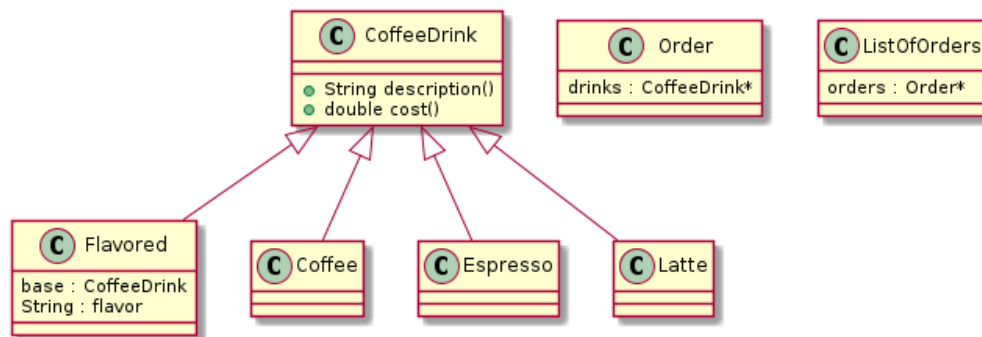
Think about the high-level structure of this program. It is something like:

```

while not done
  mainMenu.match()
  "Order":
    while not placed
      drinkMenu.match()
      "Coffee", "Espresso", "Latte":
        while not finished
          flavorMenu.match()
          "Flavor":
            Get flavor
            Decorate current drink with flavor
          "Finish":
            Add current drink to order
            finished = true
          "Cancel":
            finished = true
        "Place":
          Add current order to list
          Print current order with descriptions
          placed = true
        "Cancel":
          placed = true
      "List":
        Print and total list
      "Quit":
        done = true

```

The types seem to be



The `CoffeeDrink*` and `Order*` types mean that those types can hold zero or more of that type of object: you will implement the classes as *recursive linked lists*.

Input/Output

A `CoffeeDrink` should print out according to its type. An *actual* `CoffeeDrink` should return an empty string as its description. The description methods for `Coffee`, `Espresso`, and `Latte` return their type name as a string.

A `Flavored` has two fields, the `flavor` string and a base drink. The description is the description of the base drink followed by "with a shot of <flavor>". So a `Latte` with a shot of peppermint and a shot of hazelnut is described as

Latte with a shot of peppermint with a shot of hazelnut

When reading the flavor for a flavored drink, it is permitted to contain spaces. You should trim leading and trailing whitespace. An empty flavor string is ugly and makes descriptions unreadable. Choose one of these two possible solutions:

- loop, asking for a flavor, until the cashier enters a non-empty flavor name
- return to the decorate menu w/o having flavored the current drink; display a message before the menu that this happened

Design Considerations

The `CoffeeDrink` hierarchy is a collection of classes that all extend one base class. The `Flavored` class is an example of a `CoffeeDrink` decorator. (**How** do you know that a class is a decorator for a given base class?)

`Order` is a **linked-list** containing `CoffeeDrink` objects. The linked list routines must be *recursive*. **Suggestion:** write a `description` method that traverses the `Order` list, building up a string that contains the description of each drink **and its total** on separate lines along with the final total on a last line. Alternatively, you can write a `CoffeeDrink get(int index)` method to get the drink at the given index and build the string outside the class.

`Order` should, probably, have a `length` method and, perhaps, a `total` method.

`ListOfOrders` should be able to *total* all the orders, *count* all the drinks that were ordered, and return a string describing the whole list. With two orders of two flavored coffees and then three flavored espresso, the list should display something like

```
Order #1   2 drinks $  5.50
Order #2   3 drinks $ 12.00
Total      5 drinks $ 17.50
```

Align the **right** edge of the drink counts in a column (including in the total line) *and* align the **decimal points** in the dollar values. Align the dollar signs and do not use *too many* blanks.

Testing

We are at an awkward place. You have seen JUnit but have not done a lab with it, meaning it is still unexplored territory for you. This program will be tested *manually*.

You should make up a couple of short interaction sequences that test out edge conditions like printing an empty list of orders. Determine if it is possible to enter an *empty* order and program (and test) accordingly.

You will need to test out drinks with *no* flavors, *one* flavor, and some higher number(s) of flavors; orders with one, two, and more drinks; and order lists with zero, one, two, and more orders. Make sure the cancel choices do the right thing (and check that the canceled drink/order is **not** added anyway). Make sure you can add a drink or order after canceling one.

Documentation

README

Must document what you need to test and how you did it. Give enough information so that the grader can duplicate your tests.

Must include instructions on how to **compile** and how to **run** the program as submitted.

Deliverables

Submission medium: git to Gitea at `cs-devel.potsdam.edu`. `pMakingMenu` is the name of the repo. So, the repo, in the `drbcladd` repo would be: `S24-205-drbcladd/pMakingMenu.git`; the naming is important so that I can have a program download your work.