Decorator pattern program

Program description

The program itself has three types of coffees - Simple, Espresso and Mocha, each of which has a default price.

The user is able to add some sweeteners to the coffee like sugar, whip and chocolate, together with ice, soy or milk, in which case the price of the coffee will be increased, depending on the price of each of the additional supplements.

Object description

1. ICoffee – interface that defines the main methods used by products at the coffee shop – *getDescription()* and *getCost()*
2. Decorator – abstract decorator class that implements the *ICoffee* interface; it has the (abstract) methods *getDescription()*  and *getCost()* from *ICoffee*, as well as a protected member *tempCoffee* of type *ICoffee*
3. Espresso – Class responsible for espresso coffees, implements the *ICoffee* interface

* getDescription() – returns a description saying this is an espresso coffee
* getCost() – returns the cost of an espresso

1. SimpleCoffee – Class responsible for simple coffees, implements the *ICoffee* interface

* getDescription() – returns a description saying this is an simple coffee
* getCost() – returns the cost of a simple coffee

1. Mocha – Class responsible for mocha coffees, implements the *ICoffee* interface

* getDescription() – returns a description saying this is an mocha coffee
* getCost() – returns the cost of an mocha

1. Chocolate – Class responsible for the chocolate sprinkles sweetener, implements the *Decorator* decorator/abstract class

* Chocolate(ICoffee newCoffee) – class constructor that requires and stores a reference to another object that implements the *ICoffee* interface; the reference is stored in the protected variable *tempCoffee* from the *Decorator* class
* getDescription() – returns the description of the stored *tempCoffee*, with an additional string stating it has chocolate sprinkles as well attached at the end
* getCost() – returns the cost of the stored *tempCoffee*, as well as an additional cost equal to the price of the chocolate sprinkles

1. Whip – Class responsible for the whipped cream sweetener, implements the *Decorator* decorator/abstract class

* Whip(ICoffee newCoffee) – class constructor that requires and stores a reference to another object that implements the *ICoffee* interface; the reference is stored in the protected variable *tempCoffee* from the *Decorator* class
* getDescription() – returns the description of the stored *tempCoffee*, with an additional string stating it has whipped cream as well attached at the end
* getCost() – returns the cost of the stored *tempCoffee*, as well as an additional cost equal to the price of the whipped cream

1. Sugar – Class responsible for the sugar sweetener, implements the *Decorator* decorator/abstract class

* Sugar (ICoffee newCoffee) – class constructor that requires and stores a reference to another object that implements the *ICoffee* interface; the reference is stored in the protected variable *tempCoffee* from the *Decorator* class
* getDescription() – returns the description of the stored *tempCoffee*, with an additional string stating it has sugar as well attached at the end
* getCost() – returns the cost of the stored *tempCoffee*, as well as an additional cost equal to the price of the sugar

1. Milk – Class responsible for the milk additive, implements the *Decorator* decorator/abstract class

* Milk (ICoffee newCoffee) – class constructor that requires and stores a reference to another object that implements the *ICoffee* interface; the reference is stored in the protected variable *tempCoffee* from the *Decorator* class
* getDescription() – returns the description of the stored *tempCoffee*, with an additional string stating it has milk as well attached at the end
* getCost() – returns the cost of the stored *tempCoffee*, as well as an additional cost equal to the price of the milk

1. Soy – Class responsible for the soy additive, implements the *Decorator* decorator/abstract class

* Soy (ICoffee newCoffee) – class constructor that requires and stores a reference to another object that implements the *ICoffee* interface; the reference is stored in the protected variable *tempCoffee* from the *Decorator* class
* getDescription() – returns the description of the stored *tempCoffee*, with an additional string stating it has soy as well attached at the end
* getCost() – returns the cost of the stored *tempCoffee*, as well as an additional cost equal to the price of the soy

1. Ice – Class responsible for the ice additive, implements the *Decorator* decorator/abstract class

* Ice (ICoffee newCoffee) – class constructor that requires and stores a reference to another object that implements the *ICoffee* interface; the reference is stored in the protected variable *tempCoffee* from the *Decorator* class
* getDescription() – returns the description of the stored *tempCoffee*, with an additional string stating it has ice as well attached at the end
* getCost() – returns the cost of the stored *tempCoffee*, as well as an additional cost equal to the price of the ice

1. Form1 – the main form of the program; It stores and handles the creation and “sale” of different coffee types;

* ICoffee currentCoffee – a variable that stores an object which implements the *ICoffee* interface, used to store the current coffee being created by the program
* UpdateSelection() – Displays a string describing the currently selected coffee and its cost in the *currentCoffeeTb* text box in the form; If *currentCoffee* is null, the text box is cleared instead
* selectSimple\_Click() – sets *currentCoffee* to a new instance of the *SimpleCoffee* type and updates the description of the current selection in the form (in the *currentCoffeeTb* text box) using the *UpdateSelection()* method
* selectEspresso\_Click() – sets *currentCoffee* to a new instance of the *Espresso* type and updates the description of the current selection in the form (in the *currentCoffeeTb* text box) using the *UpdateSelection()* method
* selectMocha\_Click() – sets *currentCoffee* to a new instance of the *Mocha* type and updates the description of the current selection in the form (in the *currentCoffeeTb* text box) using the *UpdateSelection()* method
* addSugar\_Click() – if there is a coffee currently selected (*currentCoffee* is not null), sets *currentCoffee* to a new instance of the *Sugar* type, passing its constructor the current *currentCoffee object*; if there is no coffee selected, it shows a message box notifying the user
* addChocolate\_Click() – if there is a coffee currently selected (*currentCoffee* is not null), sets *currentCoffee* to a new instance of the *Sugar* type, passing its constructor the current *currentCoffee object*; if there is no coffee selected, it shows a message box notifying the user
* addWhip\_Click() – if there is a coffee currently selected (*currentCoffee* is not null), sets *currentCoffee* to a new instance of the *Whip* type, passing its constructor the current *currentCoffee object*; if there is no coffee selected, it shows a message box notifying the user
* addIce\_Click() – if there is a coffee currently selected (*currentCoffee* is not null), sets *currentCoffee* to a new instance of the *Chocolate* type, passing its constructor the current *currentCoffee object*; if there is no coffee selected, it shows a message box notifying the user
* addSoy\_Click() – if there is a coffee currently selected (*currentCoffee* is not null), sets *currentCoffee* to a new instance of the *Soy* type, passing its constructor the current *currentCoffee object*; if there is no coffee selected, it shows a message box notifying the user
* addMilk\_Click() – if there is a coffee currently selected (*currentCoffee* is not null), sets *currentCoffee* to a new instance of the *Milk* type, passing its constructor the current *currentCoffee object*; if there is no coffee selected, it shows a message box notifying the user
* finishCreation\_Click() – “finishes the sale” of the currently selected coffee, displaying the sold coffee in the *beveragesSoldLb* list box in the form and clearing the *currentCoffee* instance; if there is no *currentCoffee* instance it instead displays a message box notifying the user

3 problems

Reusability – The types of the coffees with their prices and sweeteners can be changed very easily, because of the structure of the application, including interface and decorator, which makes reusability fast and simple.

Maintainability – The constructors, the methods and the implementation at all is not that complicated with loops, serializable stuff and more, so maintaining similar programs should not be hard for the developers.

Extendibility – Basically each application can be extended with more functions. For instance in our example, more coffees can be added, with some description. It is a good idea if the coffees have some pictures, how they look like. And it will be probably nice if the user might order more than one coffee.