# STRATEGY PATTERN

# (Almost) Everything You Wanted to Know About the Open-Closed Principle

(But Were Afraid to Ask)

# Exercise

A **customer** is placed in either of three **categories**:

- **Bronze** customers: pay the full price on every item bought, plus shipping costs.
- **Silver** customers: get 10% reduction on every item bought, plus shipping costs.
- **Gold** customers: get 20 % reduction on every item bought, and free shipping.

[Tornhill 2015]

# Exercise

Write a program that calculates the final price of an item, taking into account the customer's category

**Input:**

- name and type of customer (gold, silver, bronze)
- price of the purchased good

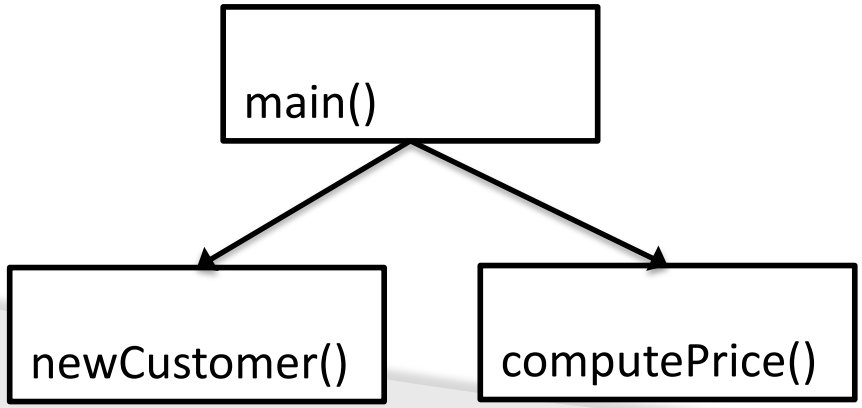
**Output:**

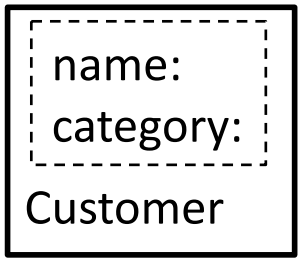
- final price

# Exercise

**Requirements**

- use a **data structure** to store the user's essential data (at least the **name** and **category**).
- the **`newCustomer()`** function will take care of the Customer data input and make the appropriate data structure
- the calculation of the final cost must be performed by the function **`computePrice()`**



# Exercise

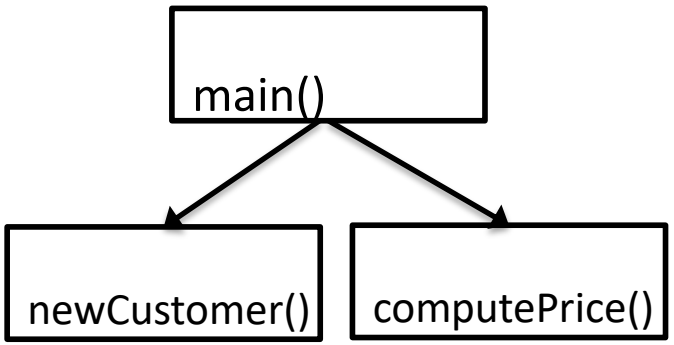
```
int main() {
 //INIT

 //INPUT and create CUSTOMER (using newCustomer() )

 //INPUT initial price

 //compute final price (using computePrice() )

 // OUTPUT
}
```

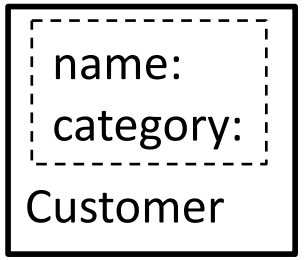


# Exercise (hint)

There are (at least) three ways to create Customer

```
struct Customer {
      char name[DIM]; //
      char* typeOfCustomer;
      // gold, silver, bronze, etc

      // to implement:
      //   List orders; address;
      //   telephone number; etc
};
```



# Exercise (hint)

1) **`main()`** creates the data structure and passes its pointer to **`newCustomer()`**

```
struct Customer c, *p_c;
p_c=&c;
p_c= newCustomer(p_c);
```

**`newCustomer()`** fills the data structure and return the pointer to the main

```
struct Customer *newCustomer(struct Customer * p_c){
  fgets(p_c->name, DIM, stdin);
  p_c->typeOfCustomer = "gold";
  …
    return p_c;
}
```

# Exercise (hint)

2) **`newCustomer()`** creates the data structure and return the data structure to the **`main()`**

```
struct Customer c;
c= newCustomer();
```

```
struct Customer newCustomer(){
   struct Customer c;
   fgets(c.name, DIM, stdin);
   c.typeOfCustomer = "gold";
  …
   return c;
}
```

# Exercise (hint)

3) **`newCustomer()`** creates the data structure and return *its pointer* to the **`main()`**

```
struct Customer *p_c;
p_c= newCustomer();
```

```
struct Customer *newCustomer(){
   // dynamic memory allocation using malloc()
   …
   fgets(p_c->name, DIM, stdin);
   p_c->typeOfCustomer = "gold";
  …
   return p_c;
}
```

# Exercise (hint)

All three of these solutions to create Customer are fine, **provided they are used and implemented correctly**.

In the proposed solution I'll use the first (but this is **not** the fundamental point of the exercise).

1) **`main()`** creates the data structure and passes its pointer to **`newCustomer()`**

```
struct Customer c, *p_c;
// or T_customer c, *p_c
// using typedef to create the alias T_customer
p_c=&c;
p_c= newCustomer(p_c);
```

# Exercise (hint)

Use 3 files:

**main.c, customerModule.c customerModule.h**

customerModule.h contain also the definition

```
typedef struct Customer {
    char name[DIM]; //
    char* typeOfCustomer; // gold, silver, bronze, etc
    // to implement:
    //      List orders; address; telephone number; etc
    //      ...
}T_customer;
```

(*this is not perfect as regards the encapsulation, but is enough for now*)

# Exercise (hint)

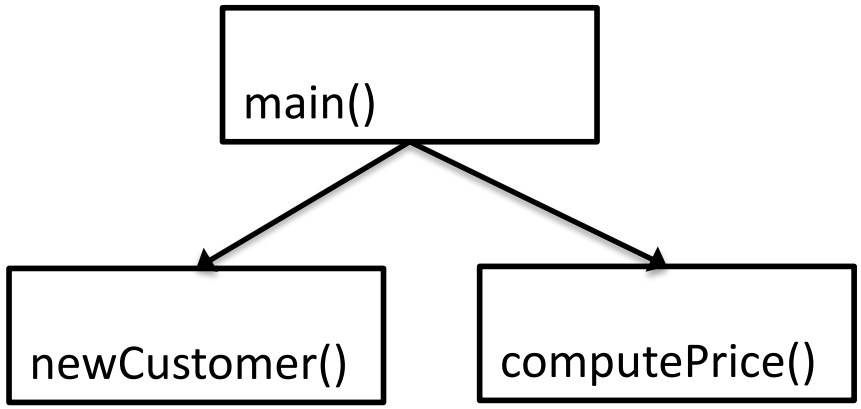
Use

strcmp(str1,str2 )

(from <string.h> )

to compare 2 strings. It returns 0 if two strings are identical.

# Exercise




Solve the exercise!

https://codeboard.io/projects/139646

Solution in https://github.com/DIEE-PAIS-code/customer_strategy_01_c

Drawbacks ?

# Exercise (refactoring)

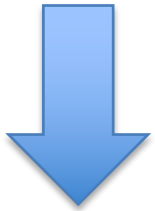
Using function to compute price in the 'if' structure

```
if (strcmp(type,"gold")==0){
    price=0.8*totalAmount;// free shipping;
```

# Exercise (refactoring)

Using function to compute price in the 'if' structure

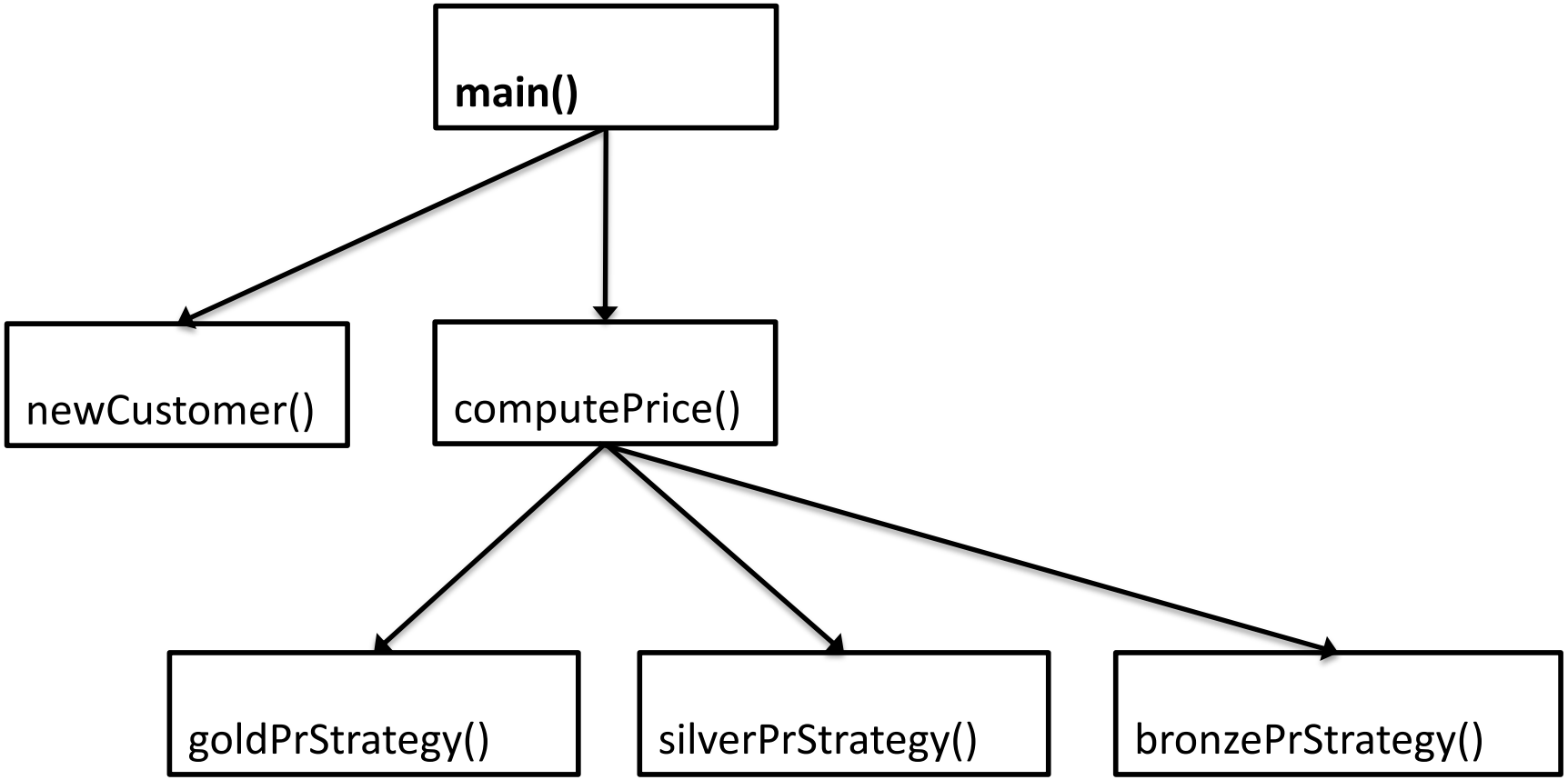
```
if (strcmp(type,"gold")==0){
    price=0.8*totalAmount;// free shipping;
```



```
if (strcmp(type,"gold")==0){
price= goldPriceStrategy(amount, shipping);
```

```
float goldPriceStrategy(float amount,
                        float shipping){
    return 0.8*amount; // free shipping;
}
```

# Exercise (discussion)



# Exercise (extension)

Add a **`greetings()`** function to greet the user
The function sends a gift card if the user is silver (10 euros) or gold (20 euros)

(it is sufficient to simulate the operations with a print)

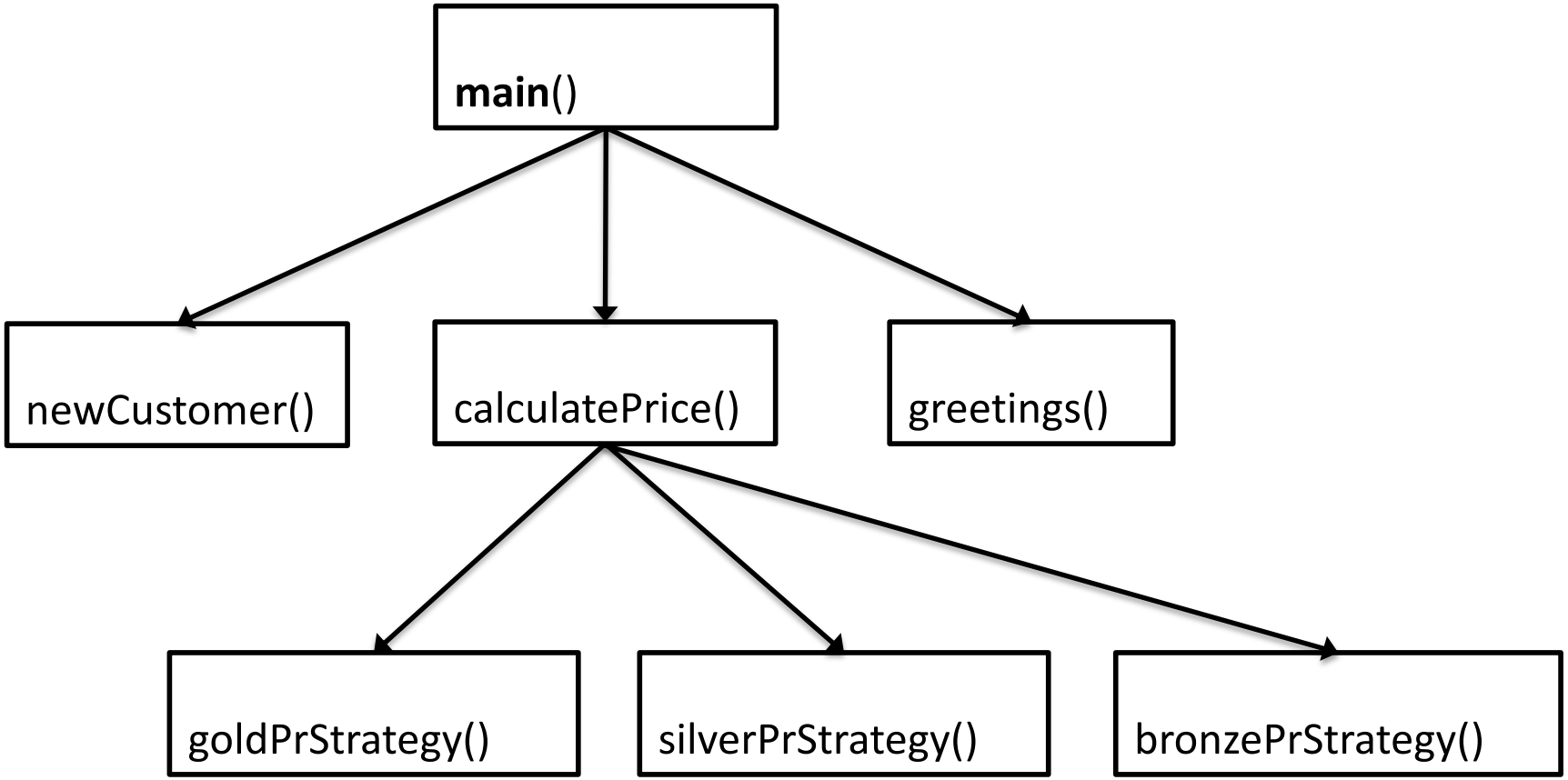
Solve the exercise!

https://codeboard.io/projects/139647

Solution in https://github.com/DIEE-PAIS-code/customer_strategy_02_c

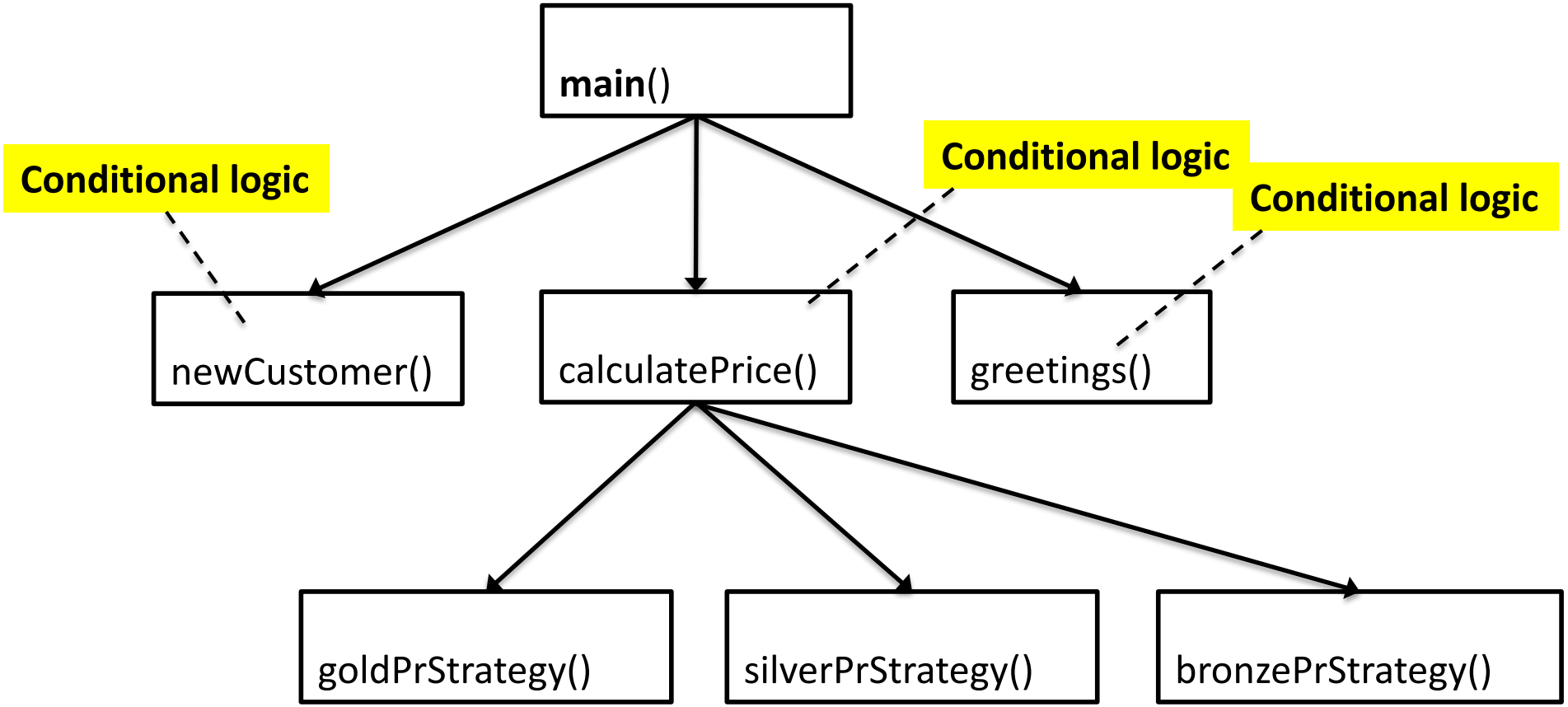
Drawbacks ?

# Exercise (discussion)




Drawbacks ?

# Exercise (discussion)




Drawbacks ?

# Discussion and Issues

1. **Conditional logic.**

Potentially complicated, nested conditional logic with k branch in n different modules.

**Risk of introducing a maintenance challenge**

D3

# Discussion and Issues

2. **Coupling**.

Remember that

the coupling characterizes the relationship (eg, interdependence) of the module with other modules. **Two modules have a high coupling if they are strictly dependent on each other.**

A low level of coupling (or loose coupling) makes it possible to analyze, understand, modify, test or reuse each module separately.

**The worst form of coupling is control coupling.**

The *client* passes a flag (category, in this case) used to control the behaviour and the inner logic of the *server* module.

**Loss of encapsulation: the client knows about the server's internals.**

# Discussion and Issues

3. **It doesn't scale.**

What happens if the **customer categories are extended**?

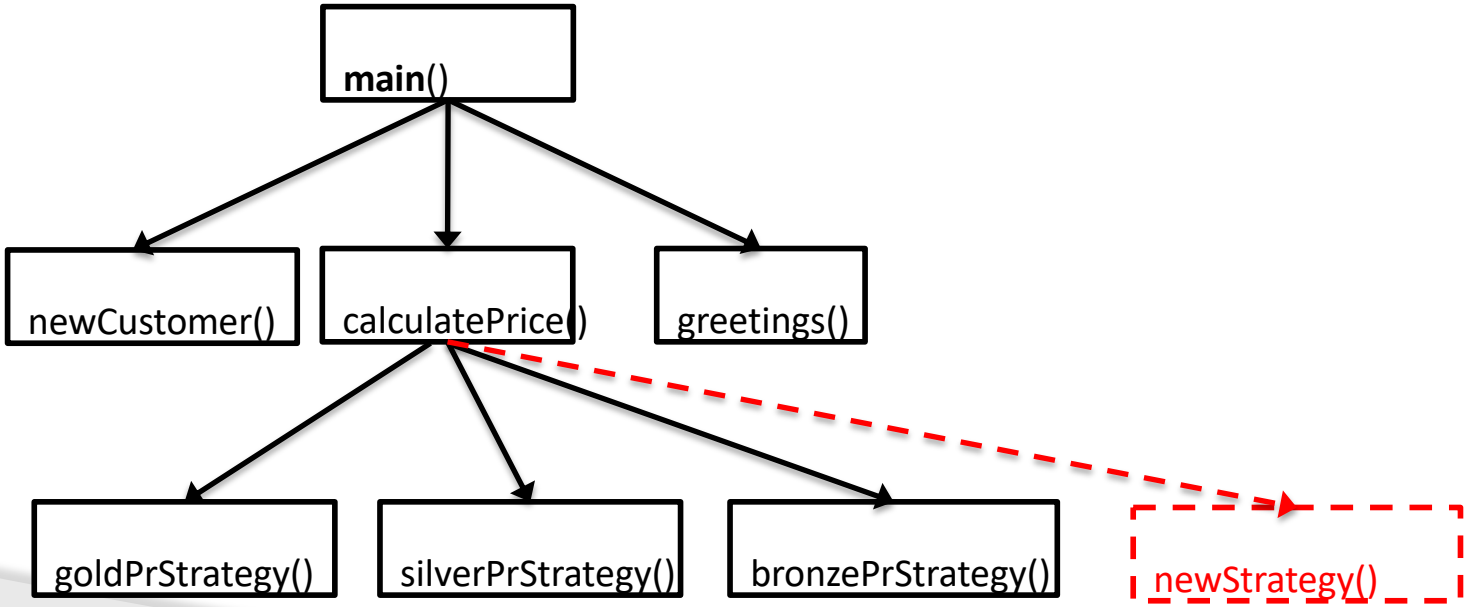
What if other **(new) modules** must apply a similar logic?

Modifying existing code is often an error-prone activity.

# The Open-Closed Principle

"***Software entities*** **should be open for extension, but closed for modification**" [Martin 2002].

Extending the behaviour is achieved by **adding entities instead of changing the existing source**.



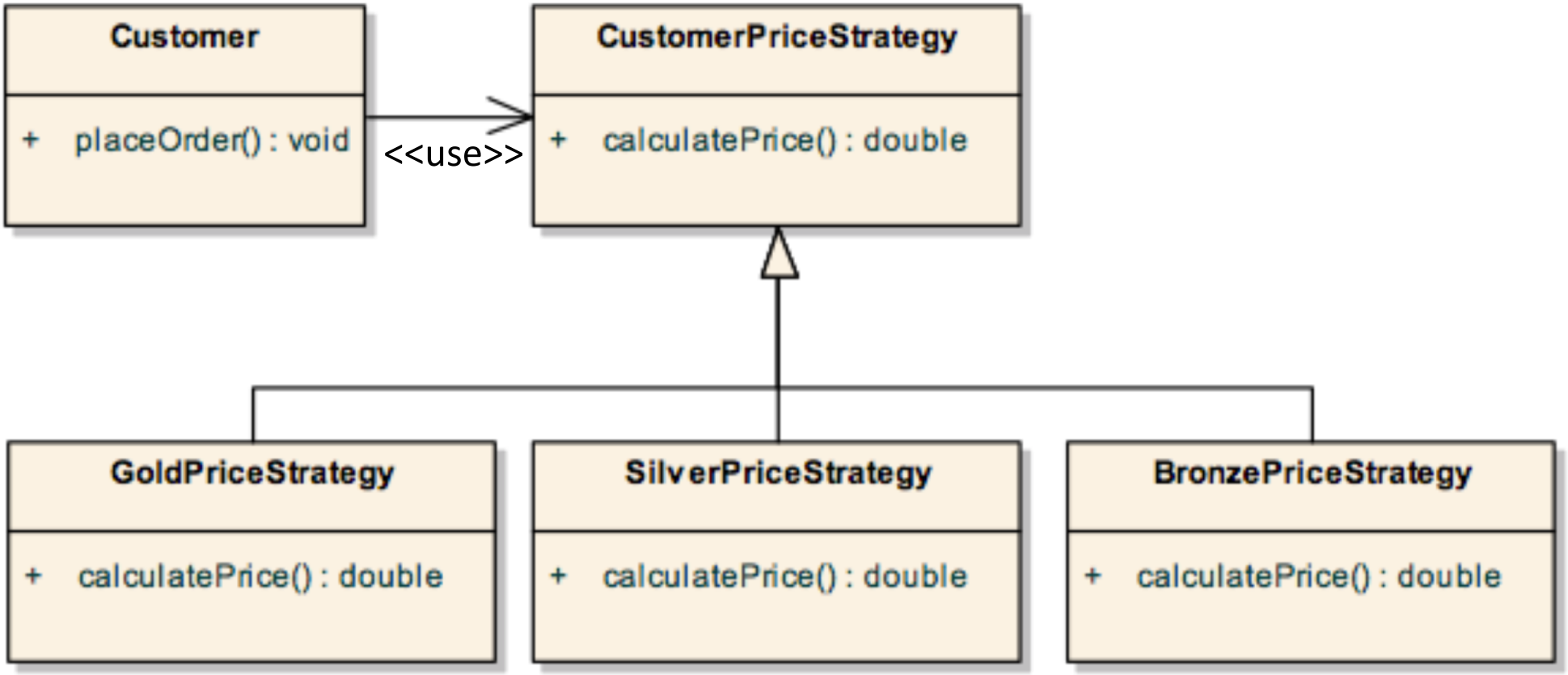
# Design pattern STRATEGY

Design pattern STRATEGY provides a way to follow and reap the benefits of the open-closed principle.

**“Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it”**

[GoF, "Design Patterns: Elements of Reusable Object-Oriented Software"]

# Design pattern STRATEGY

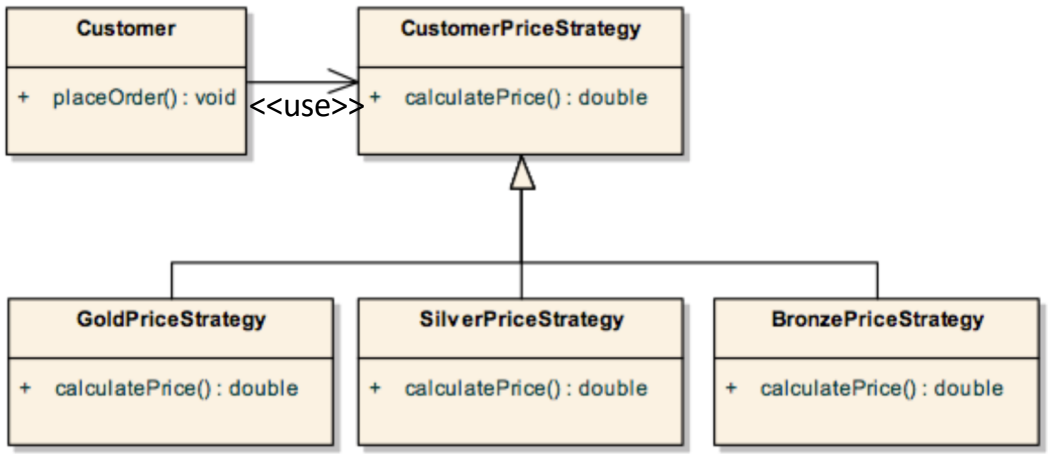


# Design pattern STRATEGY

The context (i.e. the Customer) has no knowledge about the concrete categories.

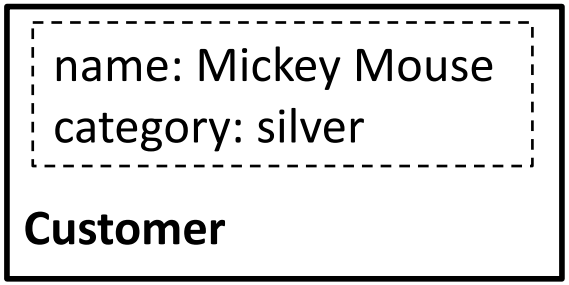
The concrete strategy is typically assigned to the context by the application. All price calculations are delegated to the assigned strategy.

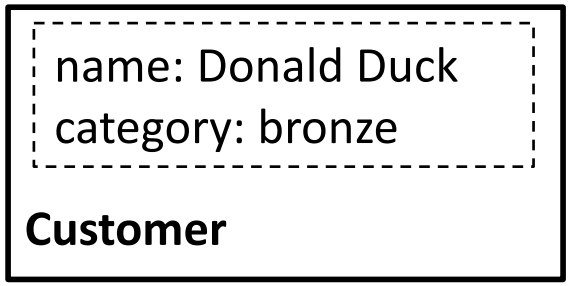
In this way it is possible to add or remove categories (i.e. price strategy)



# Solution

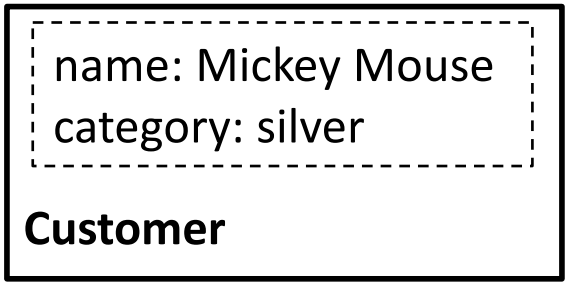
**Incorporate Customer behaviour in the data structure itself.**

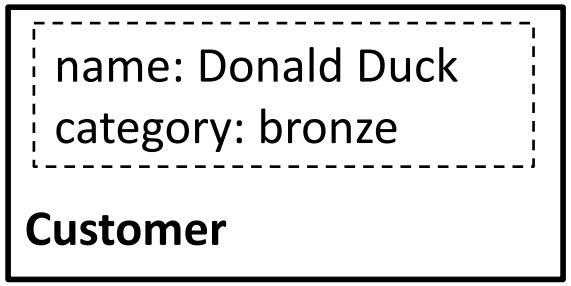


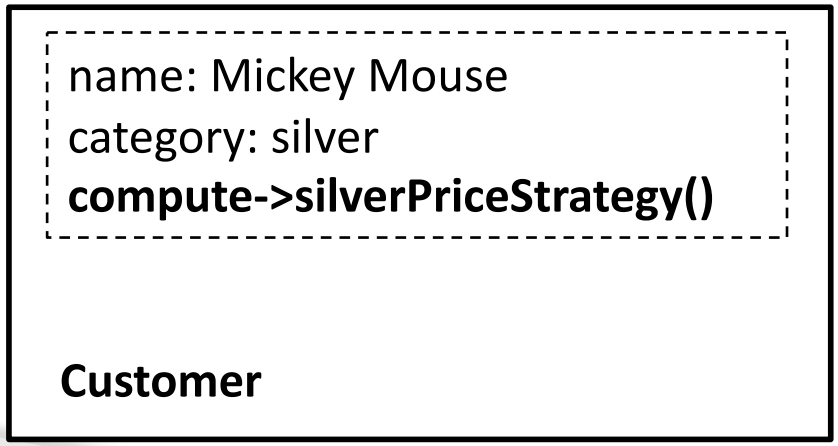



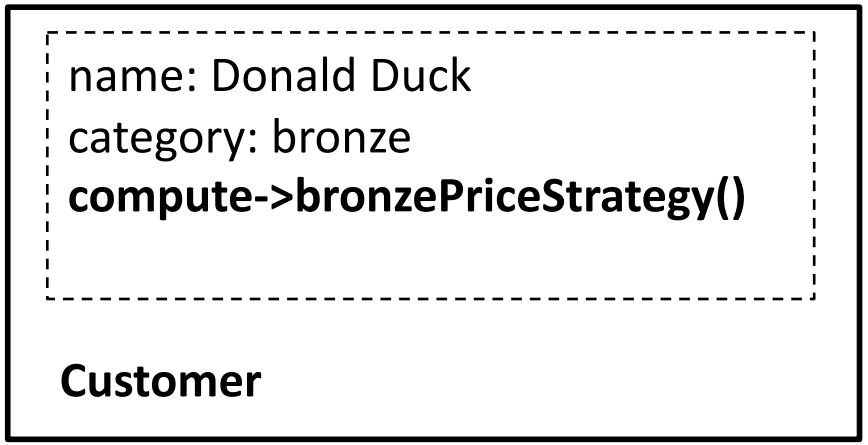
# Solution

**Incorporate Customer behaviour in the data structure itself.**



# Solution

**Incorporate Customer behaviour in the data structure itself.**

How to implement it ?

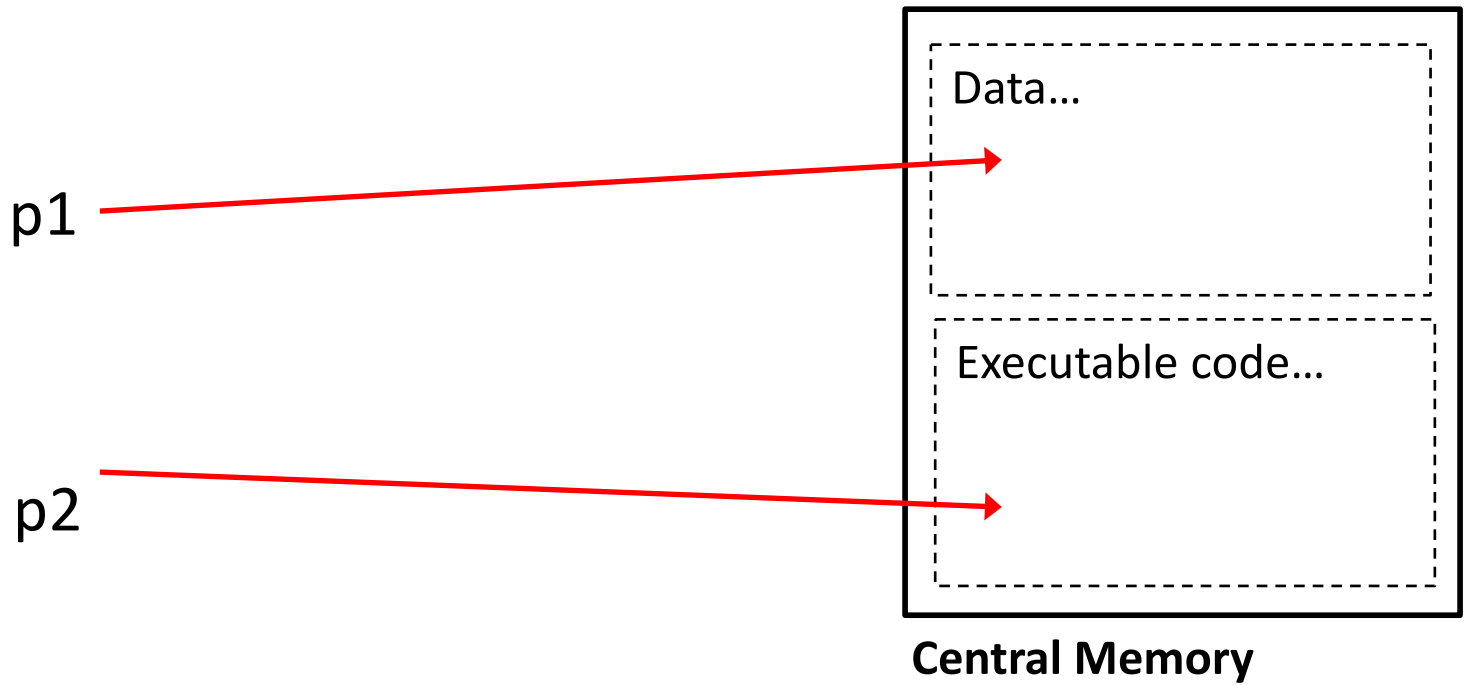
**NB!**

The discussion about the *strategy design pattern* and the problems examined above are entirely **general** and <u>do not depend on the programming language chosen</u>.
The **particular implementation** of the strategy pattern shown below, **using pointers to functions**, is relative to the c programming language. <u>But the strategy design pattern is a general design scheme</u>.

# Solution

**Incorporate Customer behaviour in the data structure itself.**

How to implement it ?  (in c)




Memory contains both **instructions and data**

# Solution

Pointers to **function**

A function pointer is a variable that stores the address of a function.

```
// function
int sum(int a, int b);

// pointer to a function
int (*p_sum)(int a, int b);
```

# Solution

Pointers to **function**

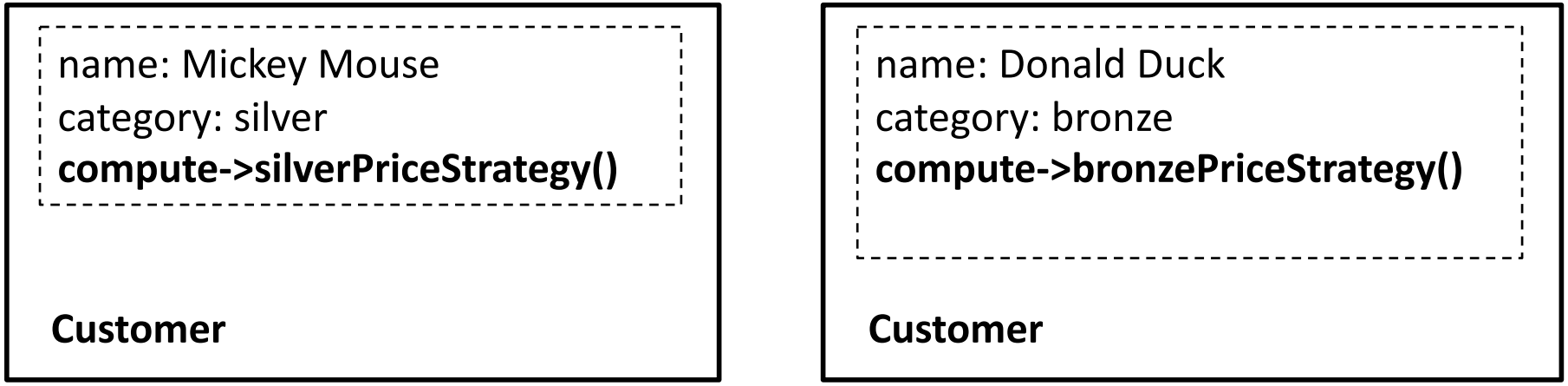
A function pointer is a variable that stores the address of a function.

```
int sum(int a, int b);
int (*p_sum)(int a, int b);

//the pointer points to the function
p_sum =&sum; //p_sum=sum;
//call
s=(*p_sum)(c1,c2); //s=p_sum(c1,c2);
```

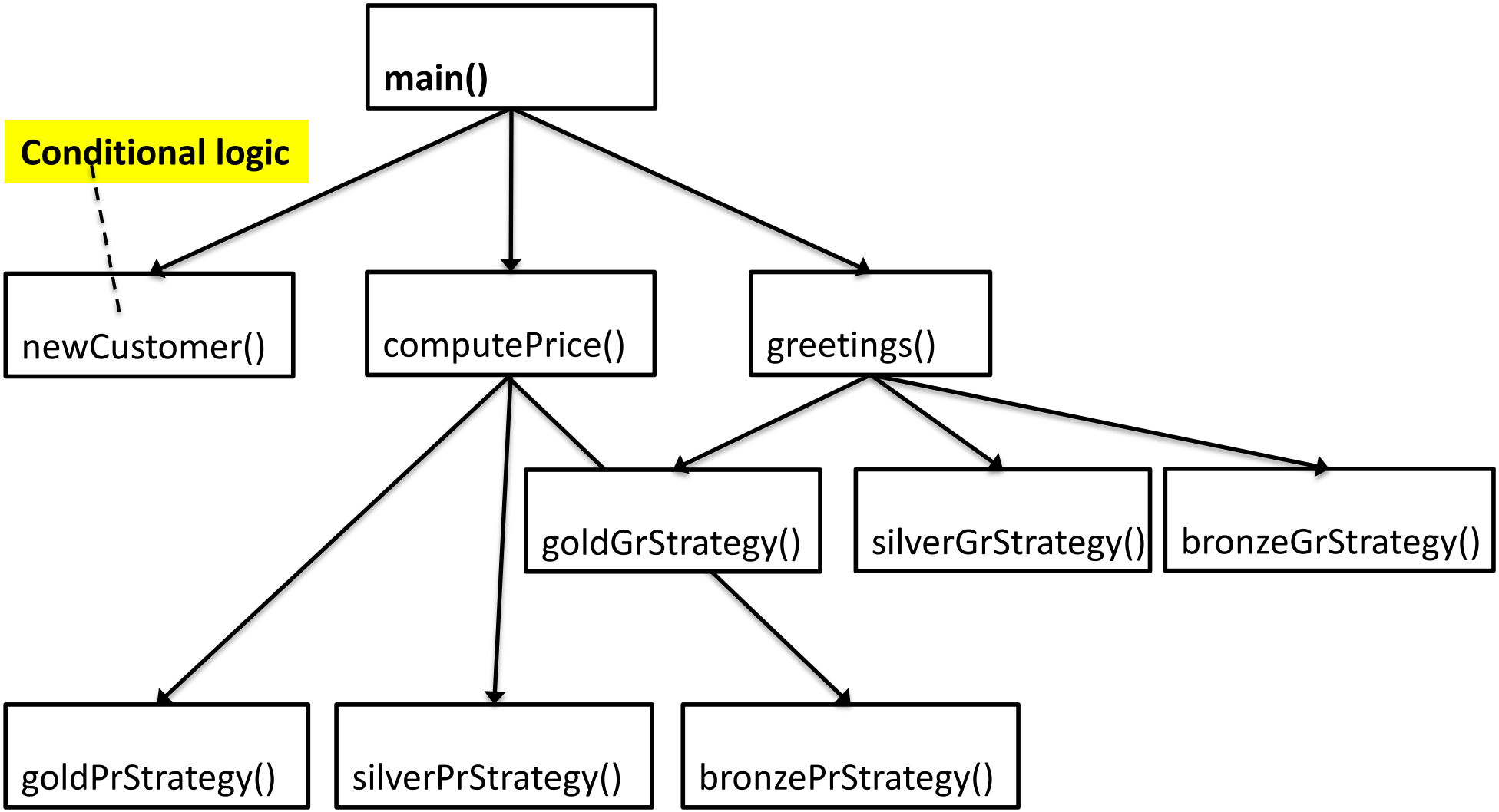
# Solution

The code now delegates the calculation to the 'strategy' (i.e. the function) associated with the customer.




```
switch (getchar()) {
    case 'a':
        p_c->typeOfCustomer = "gold";
        p_c->p_priceStrategy=goldPriceStrategy;
```

# Solution




Solution in https://github.com/DIEE-PAIS-code/customer_strategy_03_c

# Advantages

1. The benefits of the open-closed principle.

   New behaviour is supported by **adding new code** instead of modifying existing code

2. Reduces complex, conditional logic. The conditional logic compares only in one module.
3. Allows different versions of the same algorithm
4. Using the technique of pointers to functions, a run-time change of strategy simply implies pointing to another function.

# References

[Tornhill 2015] Adam Tornhill. “Patterns in C.” https://leanpub.com/patternsinc

[Martin 2002] Martin. "Agile Software Development: Principles, Patterns, and Practices"

[Gamma 1994 ] Gamma et al., "Design Patterns: Elements of Reusable Object-Oriented Software"