Patterns in C - Part 3: STRATEGY

By Adam Petersen <adampetersen75@yahoo.se>

Identifying and exploiting commonality is fundamental to software design. By encapsulating and re-using common functionality, the quality of the design rises above code duplication and dreaded anti-patterns like copy-paste. This part of the series will investigate a design pattern that adds flexibility to common software entities by letting clients customize and extend them without modifying existing code.

Control coupled customers

To attract and keep customers, many companies offer some kind of bonus system. In the example used in this article, a customer is placed in either of three categories:

- *Bronze customers*: Get 2 % reduction on every item bought.
- *Silver customers*: Get 5 % reduction on every item bought.
- *Gold customers*: Get 10 % off on all items and free shipping.

A simple and straightforward way to implement a price calculation based upon this bonus system is through conditional logic.

Listing 1: Solution using conditional logic

```
typedef enum
{
   bronzeCustomer,
   silverCustomer,
   goldCustomer
} CustomerCategory;

double calculatePrice(CustomerCategory category, double totalAmount,
                      double shipping)
{
   double price = 0;
    /* Calculate the total price depending on customer category. */
    switch(category) {
     case bronzeCustomer:
           price = totalAmount * 0.98 + shipping;
           break;
     case silverCustomer:
           price = totalAmount * 0.95 + shipping;
           break;
     case goldCustomer:
           /* Free shipping for goldcustomers.*/
           price = totalAmount * 0.90;
           break;
     default: onError("Unsupported category"); break;
    }
    return price;
}
```

Before further inspection of the design, I would like to point out that representing currency as **`double`** may lead to marginally inaccurate results. Carelessly handled, they may turn out to be fatal to, for example, a banking system. Further, security issues like a possible overflow should never be ignored in business code. However, such issues have been deliberately left out of the example in order not to lose the focus on the problem in the scope of this article.

Returning to the code, even in this simplified example, it is possible to identify three serious design problems with the approach, that wouldn't stand up well in the real-world:

1. *Conditional logic*. The solution above basically switches on a type code, leading to the risk of introducing a maintenance challenge. For example, the above mentioned security issues have to be addressed on more than one branch leading to potentially complicated, nested conditional logic.

2. *Coupling*. The client of the function above passes a flag (category) used to control the inner logic of the function. Page-Jones defines this kind of coupling as "control coupling" and he concludes that control coupling itself isn't the problem, but that it "*often indicates the presence of other, more gruesome, design ills*" [5]. One such design ill is the loss of encapsulation; the client of the function above knows about its internals. That is, the knowledge of the different customer categories is spread among several modules.

3. *It doesn't scale*. As a consequence of the design ills identified above, the solution has a scalability problem. In case the customer categories are extended, the inflexibility of the design forces modification to the function above. Modifying existing code is often an error-prone activity.

The potential problems listed above, may be derived from the failure of adhering to one, important design principle.

The Open-Closed Principle

Although mainly seen in the context of object oriented literature, the open-closed principle defines properties attractive in the context of C too. The principle is summarized as *"Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"* [1].

According to the open-closed principle, extending the behaviour of an ideal module is achieved by adding code instead of changing the existing source. Following this principle not only minimizes the risk of introducing bugs in existing, tested code but also typically raises the quality of the design by introducing loose coupling. Unfortunately, it is virtually impossible to design a module in such a way that it is closed against all kinds of changes. Even trying to design software in such a way would overcomplicate the design far beyond suitability. Identifying the modules to close, and the changes to close them against, requires experience and a good understanding of the problem domain.

In the example used in this article, it would be suitable to close the customer module against changes to the customer categories. Identifying a pattern that lets us redesign the code above in this respect seems like an attractive idea.

STRATEGY

The design pattern STRATEGY provides a way to follow and reap the benefits of the open-closed principle. *Design Patterns* [2] defines the intent of STRATEGY as "*Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it*". Related to the discussion above, the price calculations in the different customer categories are that "family of algorithms". By applying the STRATEGY pattern, each one of them gets fully encapsulated rendering the structure in Figure 1.

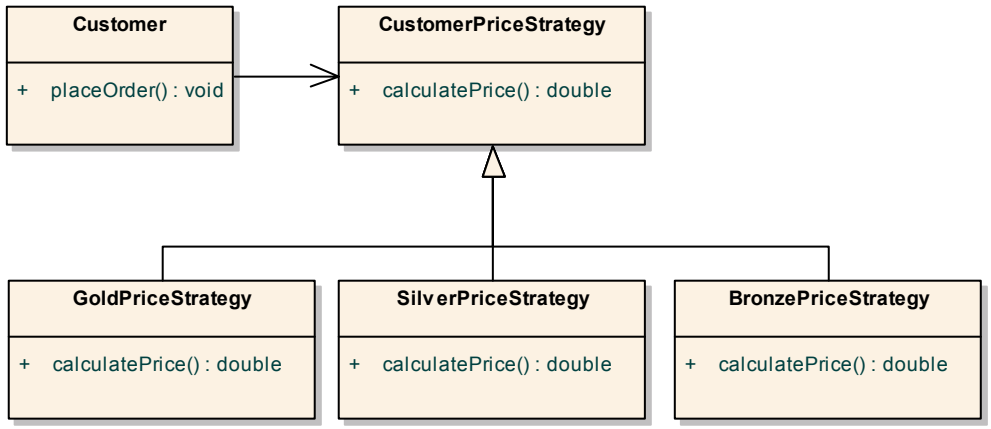

Figure 1: STRATEGY pattern structure

The context, in this example the **`Customer`**, does not have any knowledge about the concrete categories anymore; the concrete strategy is typically assigned to the context by the application. All price calculations are delegated to the assigned strategy.

Using this pattern, the interface for calculating price adheres to the open closed principle; it is possible to add or remove categories without modifying the existing calculation algorithms or the **`Customer`** itself.

Implementation Mechanism

When implementing the STRATEGY pattern in C, without language support for polymorphism and inheritance, an alternative to the object oriented features has to be found for the abstraction. This problem is almost identical to the one faced when implementing the STATE pattern [4], indicating that the same mechanism may be used, namely pointers to functions. The possibility to specify pointers to functions proves to be an efficient technique for implementing dynamic behaviour.

In a C-implementation, the basic structure in the diagram above remains, but the interface is expressed as a declaration of a pointer to function.

Listing 2: Strategy interface in `CustomerStrategy.h`

```
typedef double (*CustomerPriceStrategy)(double amount, double shipping);
```

The different strategies are realized as functions following the signature specified by the interface. The privileges of each customer category, the concept that varies, are now encapsulated within their concrete strategy. Depending on the complexity and the cohesion of the concrete strategies, the source code may be organized as either one strategy per compilation unit or by gathering all supported strategies in a single compilation unit. For the simple strategies used in this example, the latter approach has been chosen.

Listing 3: Interface to the concrete customer categories in `CustomerCategories.h`

```
double bronzePriceStrategy(double amount, double shipping);

double silverPriceStrategy(double amount, double shipping);

double goldPriceStrategy(double amount, double shipping);
```

Listing 4: Implementation of the concrete customer strategies in `CustomerCategories.c`

```
/* In production code, all input should be validated and the calculations secured
upon entry of each function. */
double bronzePriceStrategy(double amount, double shipping)
{
   return amount * 0.98 + shipping;
}

double silverPriceStrategy(double amount, double shipping)
{
   return amount * 0.95 + shipping;
}

double goldPriceStrategy(double amount, double shipping)
{
   /* Free shipping for gold customers. */
   return amount * 0.90;
}
```

Using the strategies, the context code now delegates the calculation to the strategy associated with the customer.

Listing 5: `Customer.c`

```
#include "CustomerStrategy.h"
/* Other include files omitted... */

struct Customer
{
   const char* name;
   Address address;
   List orders;
   /* Bind the strategy to Customer. */
   CustomerPriceStrategy priceStrategy;
};

void placeOrder(struct Customer* customer, const Order* order)
{
   double totalAmount = customer->priceStrategy(order->amount, order->shipping);
   /* More code to process the order… */
}
```

The code above solves the detected design ills. As the customer now only depends upon the strategy interface, categories can be added or removed without changing the code of the customer and without the risk of introducing bugs into existing strategies. The remaining open issue with the implementation is to specify how a certain strategy gets bound to the customer.

Binding the Strategy

The strategy may be supplied by the client of the customer and bound to the customer during its creation, or an initial strategy may be chosen by the customer itself. The former alternative is clearly more flexible as it avoids the need for the customer to depend upon a concrete strategy. The code below illustrates this approach, using a customer implemented as a FIRST-CLASS ADT [3].

Listing 6: Binding strategy upon creation, `Customer.c`

```
CustomerPtr createCustomer(const char* name, const Address* address,
                           CustomerPriceStrategy priceStrategy)
{
   CustomerPtr customer = malloc(sizeof *customer);

   if(NULL != customer){
      /* Bind the initial strategy supplied by the client. */
      customer->priceStrategy = priceStrategy;

      /* Initialize the other attributes of the customer here. */
   }

   return customer;
}
```

Listing 7: Client code specifying the binding

```
#include "Customer.h"
#include "CustomerCategories.h"

static CustomerPtr createBronzeCustomer(const char* name, const Address* address)
{
   return createCustomer(name, address, bronzePriceStrategy);
}
```

Depending on the problem at hand, a context may be re-bound to another strategy. For example, as a customer gets promoted to the silver category, that customer should get associated with the **`silverPriceStrategy`**. Using the technique of pointers to functions, a run-time change of strategy simply implies pointing to another function.

Listing 8: Rebinding a strategy, `Customer.c`

```
void changePriceCategory(CustomerPtr customer,
                         CustomerPriceStrategy newPriceStrategy)
{
   assert(NULL != customer);
   customer->priceStrategy = newPriceStrategy;
}
```

Yet another alternative is to avoid the binding altogether and let the client pass the different strategies to the context in each function invocation. This alternative may be suitable in case the context doesn't have any state memory. However, for our example, which uses first-class objects, the opposite is true and the natural abstraction is to associate the customer with a price-strategy as illustrated above.

Comparison of STRATEGY and STATE

When discussing the STRATEGY pattern, its relationship with the pattern preceding it in the *Design Patterns* [2] book deserves special mention. The design patterns STATE [2] [4] and STRATEGY are closely related. Robert C. Martin puts it this way: *"all instances of the State pattern are also instances of the Strategy pattern, but not all instances of Strategy are State"* [1].

This observation leads us to a recommendation by John Vlissides, co-author of *Design Patterns*, stating that "*Let the* **intents** *of the patterns be your guide to their differences and not the class structures that implement them*" [6]. And indeed, even though STATE and STRATEGY have a similar structure and use similar mechanisms, they differ in their intent. The STATE pattern focuses upon managing well-defined transitions between discrete states, whereas the primary purpose with STRATEGY is to vary the implementation of an algorithm.

Related to the example used in this article, had the categories been based on the sum of money spent by a customer, STATE would have been a better choice; the categories would basically illustrate the lifecycle of a customer, as the transitions between categories depend upon the history of the customer object itself. The STATE pattern may be used to implement this behaviour as a finite state machine.

On the other hand, in case the placement in a certain category is the result of a membership fee, STRATEGY is a better abstraction. It is still possible for a customer to wander between different categories, but a transition to a new category doesn't depend upon the history of that customer object.

Although not to be taken as a universal truth, a further observation relates to the usage of these two patterns and their relationships towards the client. STATE tends to be an internal concern of the context and the existence of STATE is usually encapsulated from the client. Contrasting this with STRATEGY, the client usually binds a concrete strategy to its context.

Consequences

The main consequences of applying the STRATEGY pattern are:

1. *The benefits of the open-closed principle*. The design pattern STRATEGY offers great flexibility in that it allows clients to change and control the behavior of an existing module by implementing their own, concrete strategies. Thus, new behavior is supported by adding new code instead of modifying existing code.

2. *Reduces complex, conditional logic*. Complex interactions may lead to monolithic modules littered with conditional logic, potentially in the form of control coupling. Such code tends to be hard to maintain and extend. By encapsulating each branch of the conditionals in a strategy, the STRATEGY pattern eliminates conditional logic.

3. *Allows different versions of the same algorithm*. The non-functional requirements on a module may typically vary depending on its usage. The typical trade-off between an algorithm optimized for speed versus one optimized with respect to memory usage is classical. Using the Strategy pattern, the choice of trade-offs may be delegated to the user ("*Strategies can provide different implementations of the* **same** *behavior*" [2]).

4. *An extra level of indirection*. The main issue with this consequence arises as data from the context has to be obtained in a strategy. As all functions used as strategies have to follow the same signature, simply adding potentially unrelated parameters lowers the cohesion. Implementing the context as a FIRST-CLASS ADT [3] may solve this problem as it reduces the parameters to a single handle. With the FIRST-CLASS ADT approach, the knowledge about the data of interest is encapsulated within each strategy and obtained through the handle to the FIRST-CLASS ADT. A careful design should strive to keep the module implemented as a FIRST-CLASS ADT highly cohesive and avoid having it decay into a repository of unrelated data needed by different strategies (such a design ill typically indicates that the wrong abstraction has been chosen). Similarly, in case the flexibility of the STRATEGY pattern isn't needed and the problem may be solved by conditional logic that is easy to follow, the latter is probably a better choice.

Example of use

STRATEGY may prove useful for specifying policies in framework design. Using STRATEGY, clients may parameterize the implementation.

For example, error-handling is typically delegated to the application. Such a design may be realized by letting the client provide a strategy to be invoked upon an error. By those means, the error may be handled in an application specific manner, which may stretch between simply ignoring the error to logging it or even reboot the system.

Next time

The next article will look further into dependency management and continue to build on the open-closed principle by introducing a C implementation of the OBSERVER [2] pattern.

References

1. Robert C. Martin, "Agile Software Development", Prentice Hall
2. Gamma, E., Helm, R., Johnson, R., and Vlissides, J, "Design Patterns", Addison-Wesley
3. Adam Petersen, "Patterns in C, part 1", C Vu 17.1
4. Adam Petersen, "Patterns in C, part 2: STATE", C Vu 17.2
5. Meilir Page-Jones, "The Practical Guide to Structured Systems Design", Prentice Hall
6. John Vlissides, "Pattern Hatching", Addison-Wesley

Acknowledgements

Many thanks to Magnus Adamsson, Tord Andersson, Drago Krznaric and André Saitzkoff for their feedback.