The Cool Shop project: - Phase 1

“CoolShop” is a large reatil shopping mall which has chain of mall’s in Mumbai and Pune city. The company management wants a simple customer management system for their retail shops with the following features.

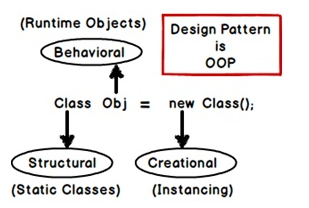
Company has decided to launch the project in phases.

So in the first phase they just want to capture the customer infomation. Below is the requirement in more details:-

1. Application would be capturing 5 fields for now Customer Name, Phone number, Bill Amount, Bill date and Customer Address
2. In phase 1 two types of customer data are collected. One is the lead and the other is a customer. A lead is a person who comes to cool shop but does not buy anything. He just enquires and goes. A customer is a person who comes and buys things from the shop. A customer actually does a financial transaction.
3. When it’s a lead only Customer name and phone number is compulsory but for a Customer all fields are compulsory. System should have a provision to add new validation rules seamlessly and these validation rules should be flexible and reusable to apply to the system.
4. For now the system will use SQL Server and ADO.NET as the data layer technology. But in the coming months we would be migrating this data layer to EF. Migration should be seamless without many changes across system.
5. System should have the ability of cancelling any modification done on the screen. So if the customer is editing a record and he has changed some values, he should have the opportunity to revert back to the old values.

OOP is a three phase process:-

* Template creation
* Instantiate
* Run

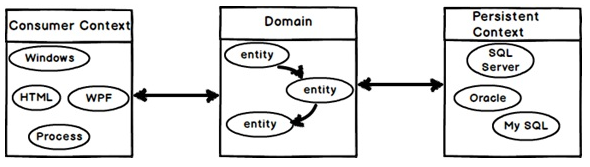


The technical context

The above defined three classes. Now two things need to be done

1. Some UI should invoke these class, bring these entities live in to RAM
2. Second once the end user finishes his operation we need to store these entities in to the hard disk

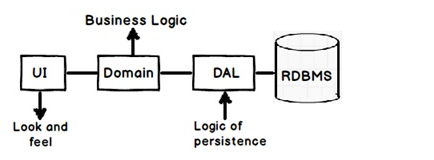
Entities need IT infracstructure to run. It needs a UI infrastructure for invoking and persistent infrastructure for saving to hard disk.



Three Lyaer architecture – Managing changes

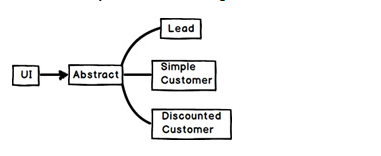
Litmus test of software architecture happens during changes. When you change in one place if you are changing in lot of places that mean it’s a sign of bad architecture.

So to avoid changes all over places we need to make proper layers and compartments and put similar nature of responsibility in thoes layers



Decoupling needs abstract thinking – Creating interfaces

In order to achieve decoupling between the UI and the Customer types, the UI has to see the Customer types in an abstract way rather than dealing with concrer classes.



The UI should be only talking with pure definitions than implemented concrete classes

Which are three main categories of design pattern.

Creational, Structural and Behavioral

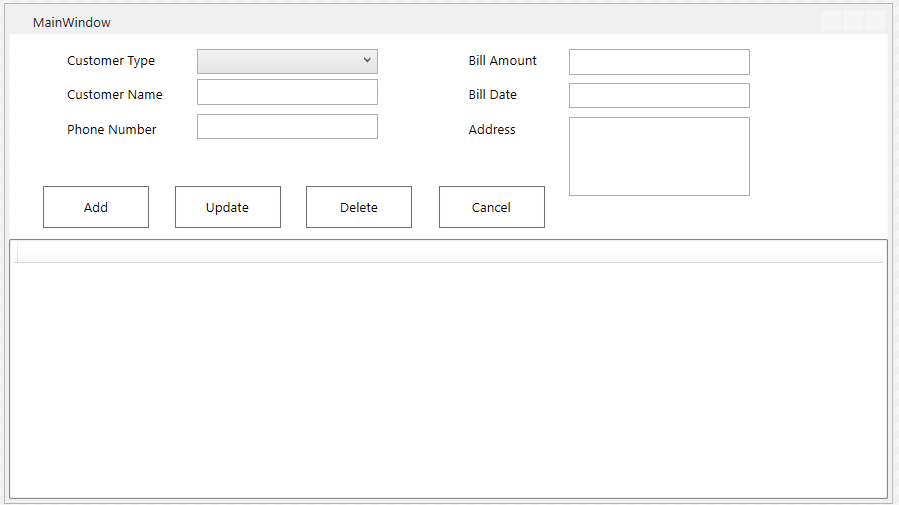
**Creational patternts**

* Abstract factory
* Builder
* Factory Method
* Prototype
* Singleton

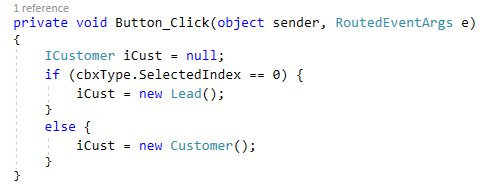
Giait thích về Factory pattern

Factory pattern là một loại trong creational patterns. Bối cảnh sinh ra pattern này là như sau.

Giả dụ ta có một giao diện



Nếu như bình thường, sau khi điền các thông tin chúng ta sẽ tiến hành Add dữ liệu. Và code behind của chúng ta sẽ như sau.

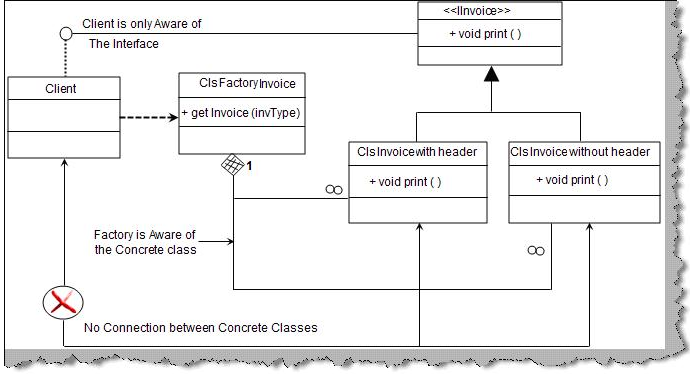


Code trên gặp vấn đề gì ?

Nếu code chỉ để chạy thì ok, chúng ta sẽ chạy ngon lành. Nghĩ xa hơn, nếu nhưng chúng ta có nhiều hơn 2 loại customer và sau này khách hàng lại sửa muốn thêm vài loại nữa thì lúc này có các vấn đề xảy ra như sau.

1. Nếu như thế này có nghĩa chúng ta sẽ phải sửa code đã có sẵn, và vi phạm nguyên lý SOLID và cụ thể ở đây chúng ta đang vi phạm nguyên lỹ chữ O – Open/Closed Principle. Nội dung của nguyên lý thứ 2 là “Objects or entities should be open for extension, but closed for modification”. Nếu như ở đây có thêm Manager thì chúng ta phải mở rộng if/else => chống lại nguyên lý số 2 về thiết kế.
2. Code của chúng ta đang vi phạm một nguyên lý thiết kế nữa đó chính là nguyên lý chữ S – Single responsibility principle. Tức là một class hay một method chỉ đảm nhiệm một vai trò, hay tương đương với nguyên lý này là SOC – Separate of concern, một class chỉ lo lắng cho một thứ. Ở đây class của chúng ta ngoài việc đảm nhiệm vai trò làm giao diện, xử lý các sự kiện, vì vậy việc kiêm thêm công tác xử lý logic, lo lắng về việc tạo một objects mới sẽ gây ra độ phức tạp rất lớn.
3. Đây đang là layer client, và chúng ta đang có quá nhiều keyword ‘new’ , nói cách khác thì client phải load quá nhiều hoạt động creation object => làm logic client phức tạp. Nếu chúng ta thêm 1 class teamplate nữa là Manager thì có nghĩa chúng ta cần reference vào trong client và complie lại.

Để giải quyết vấn đề này, chúng ta có thể chuyển giao logic tạo mới sang một class khác. Tạo một mắt xích làm kết nối chúng đó là Icustomer, khi đó client chỉ giao tiếp qua Icustomer, đây cũng là một cái phù hợp với nguyên lý IOC, đảo ngược control từ client sang một class khác. Ứng cử viên sách giá để giải quyết vấn đề này là Factory pattern



Như trên ta có thể thấy, Client không kết nối trực tiếp với concrete Invoice class. Nó sẽ kết nối với abstract là Iinvoice. Cung cấp các concrete class object là FactoryInvoice, nhận đầu vào là type và nó sẽ xử lý logic để trả ra object phù hợp dưới dạng abstract tới client.

Abstract Factory

Behavioral Design Patterns

Behavioral design patterns có liên quan đến các thuật toán và sự phân công responsibility của objects.

1. Chain of responsibility

Là desgin pattern cho phép bạn pass requests dọc theo một chuỗi các xử lý. Khi đang nhận request, mooix handler quyết định process the request hoặc pass nó thới the next handler trong chuỗi xử lý.

