**Polymorphic Behavior Through Inheritance**

**Summary:** *Trong một lớp thường, ta khai báo một method là virtual để cac thằng con có thể override lại method đó khi nó kế thừa từ thằng cha, và khi thực thi thằng con sẽ gọi đến method đó là của nó chứ không phải là thằng cha (khi nó đã override) còn không nó sẽ lấy method mặc định của thằng cha (vì nó thừa kế từ thằng cha nên tài sài của cha là của con chứ).*

Một abstract class thì không thể instantiated (tạo thực thể). Nó thường chứa property và method nhưng mấy cái này không được implementation. Mấy thằng nào kế thừa từ nó thì bắt buộc phải implement mấy thằng từ của thằng cha mà nó kế thừa.

public class Contact

{

public string FirstName { get; set; }

public string LastName { get; set; }

public string EmailAddress { get; set; }

public string PhoneNo { get; set; }

public virtual string GetDetail()

{

return FirstName + " " + LastName + " (" + EmailAddress + ", " + PhoneNo + ")";

}

}

Ta khai báo lớp Contact như sau:

Lớp contact ta thấy rằng thằng GetDetail() là virtual, mục đính là những thằng con nào kế thừa từ nó thì có thể override lại theo cách của tụi nó hoặc lấy của thằng cha ra dùng luôn nếu như hành vi GetDetail() của thằng con và thằng cha là giống nhau.

Cũng tương tự như vậy ta thêm 2 lớp con cũng kế thừa từ thằng contact vi GetDetail() của thằng con và thằng cha là giống nhau.thì có thể over

Ta viết một số thứ để test các lớp ta đã khai báo phía trên.

public class BusinessContact: Contact

{

public string CompanyName { get; set; }

public string Designation{ get; set; }

public override string GetDetail()

{

return FirstName + " " + LastName + " (" + Designation + ", " + CompanyName + ")";

}

}

public class ProfessionalContact: Contact

{

public string Service { get; set; }

public string Address { get; set; }

public string Timing { get; set; }

public override string GetDetail()

{

return FirstName + " " + LastName + " (" + Service + ", " + Timing + ")";

}

}

Khi ta thực thi đoạn chương trình trên thì mỗi thằng con sẽ show ra đúng thông tin bằng việc gọi đúng method ShowDetails mà nó đã override của thằng cha.

class Program

{

public static void ShowDetails(Contact c)

{

string details = c.GetDetail();

Console.WriteLine(details);

Console.ReadLine();

}

static void Main(string[] args)

{

BusinessContact c1 = new BusinessContact();

c1.FirstName = "Nancy";

c1.LastName = "Davolio";

c1.EmailAddress = "nancy@localhost";

c1.PhoneNo = "(206) 555-9857";

c1.CompanyName = "Northwind Traders Inc.";

c1.Designation = "Sales Representative";

ProfessionalContact c2 = new ProfessionalContact();

c2.FirstName = "Andrew";

c2.LastName = "Fuller";

c2.EmailAddress = "andrew@localhost";

c2.PhoneNo = "(206) 555-9482";

c2.Service = "Doctor";

c2.Address = "908 W. Capital Way, Tacoma, USA";

c2.Timing = "10 AM to 6 PM";

PersonalContact c3 = new PersonalContact();

c3.FirstName = "Janet";

c3.LastName = "Leverling";

c3.EmailAddress = "janet@localhost";

c3.PhoneNo = "(206) 555-3412";

c3.BirthDate = new DateTime(1971, 3, 20);

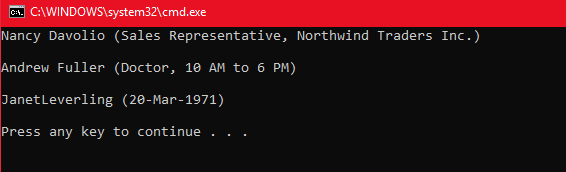
ShowDetails(c1);

ShowDetails(c2);

ShowDetails(c3);

}

}



số thứ để test các lớp ta đã khai báo phía trên.

**Polymorphic Behavior Through Interfaces**

**Summary:** *Ta có một thằng (tạm gọi là thằng cha) là một interface, ta cho các thằng con implement từ thằng này. Do vậy, thằng cha sẽ cover các thằng con, và khi implement một method hay một component nào ta khai báo kiểu thằng cha thì khi thằng con vào nó sẽ tự ép kiểu sang thằng con. Điều này dễ dàng thấy rõ được áp dụng trong design pattern tên là method factory.*

public interface ICountryTaxCaculator

{

decimal CaculateTaxAmount();

}

public class TaxCalculatorForUS : ICountryTaxCaculator

{

public decimal CaculateTaxAmount()

{

return 10000m;

}

}

public class TaxCalculatorForUK : ICountryTaxCaculator

{

public decimal CaculateTaxAmount()

{

return 20000m;

}

}

public class TaxCalculatorForIn : ICountryTaxCaculator

{

public decimal CaculateTaxAmount()

{

return 5000m;

}

}

class Program

{

public static void ShowDetail(ICountryTaxCaculator t)

{

decimal tax = t.CaculateTaxAmount();

Console.WriteLine("Tax Amount: "+tax);

Console.ReadLine();

}

static void Main(string[] args)

{

TaxCalculatorForUS tUS = new TaxCalculatorForUS();

TaxCalculatorForUK tUK = new TaxCalculatorForUK();

TaxCalculatorForIn tIN = new TaxCalculatorForIn();

ShowDetail(tUS);

ShowDetail(tUK);

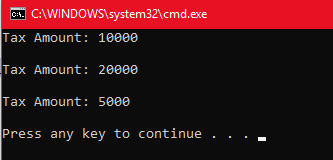
ShowDetail(tIN);

}

}

Khi ta thực thi chương trình thì khi gọi đến hàm ShowDetail() thì tùy vào thằng đối tượng nào mà nó sẽ gọi ra đúng thằng method tính thuế (CaculateTaxAmount()) của chính nó để thực hiện việc tính thuế cho nó.

Kết quả của chương trình như sau:



**Overview of SOLID Principles**

**SOLID**

**S**ingle Responsibility Principle  
**O**pen/Closed Principle  
**L**iskov Substitution Principle  
**I**nterface Segregation Principle  
Dependency Inversion Principle

Có 3 nhóm chính cho các design patterns.

1. **Creational patterns**

Factory Method  
Abstract Factory

Builder

Prototype

Singleton

1. **Structural patterns**

Adapter

Bridge

Composite

Decorator

Façade

Flyweight

Proxy

1. **Behavioral patterns**

Interpreter

Template Method

Chain of Responsibility

Command

Iterator

Mediator

Memento

Observer

State

Strategy

Visitor

1. **Single Responsibility Principle**

***A class should have only a single responsibility***

***(tạm dịch: nhà ở là để ở chứ không thể vừa là nhà ở mà vừa là toilet công cộng được)***



Ở trên ta có lớp **CustomerSearch**, lớp này cho người dùng theo **Country** hay **CompanyName** hay **ContactName** và users vô cùng hạnh phúc với các chức năng này. Một ngày đẹp trời bạn nhận được một yêu cầu là user muốn export các kết quả tìm kiếm ra file **CSV** để mà họ có thể tải về làm tài liệu du học chẳng hạn. Để làm việc này bạn thêm vào một method tên là **ExportToCSV**(data) thật là easy.



Tưởng chừng mọi thứ đã ngon lành cành đào. Người ta nói dời méo như mơ và điều đó rất là đúng. Một ngày đẹp trời bạn nhận được một yêu cầu trời đánh là user muốn export ra XML, PDF, docx hay xxxx cái quần què gì đó chứ méo muốn ra CSV nữa. Và câu chuyện bắt đầu từ đây, bạn design class **CustomerSearch** để cho tụi user nó search, đó là chức năng chính của nó. Nhưng giờ bạn lại thêm vào một chức năng là export kết quả và thế là mỗi lần thằng user muốn thay đổi export loại gì bạn phải vác xác vào sửa lại lớp **CustomerSearch** và việc này thì như là cực hình vậy, bạn sửa export thì hiển nhiên phải test lại nó rồi nhưng cũng phải test lại thằng CustomerSearch trong khi ta chả xớ mú gì đến nó. Hãy tưởng tượng điều này kiểu như bạn xây ngôi nhà để ở nhưng lại để cái toilet công cộng vào phòng khác và mỗi lần bên dịch vụ công ích muốn trang trí lại cái cầu ỉa thì phải bay vào nhà bạn đập banh chành cái phòng khách để sửa cái cầu ỉa công cộng đó thôi. Do vậy, từ vấn đề này bạn rút ra là nhà mình là chỉ là nhà mình không nên có thêm tính năng cầu ỉa công cộng nếu không hậu quả sẽ khó lường, lập trình cũng vậy, một lớp chỉ đảm nhận một mission duy nhất như lúc thiết kế cho nó vậy.

Vậy giải pháp cho vấn đề trên ta nên giải quyết thế nào:

Ta chỉ việc tách việc export ra một class khác có tên là CustomerDataExporter lúc này ta có 2 classes như sau:



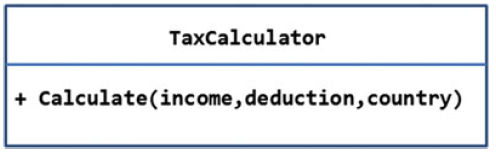
Lúc này thì 2 thằng classes là **CustomerSearch** và **CustomerDataExporter** méo còn ăn nhậu với nhau nữa, đường ai nấy đi và do vậy mỗi lần muốn sửa thằng nào thì chỉ cần test lại thằng đó chứ méo có phải test thằng khác. ***Điều này có thể phát biểu là BF của bạn thì không nền là GF của anh trai của bạn :v .***

1. **Open/Closed Principle (OCP)**

***A class should be open for extension but closed for modification .***

Nguyên lý này có thể được hiểu như sau: Bạn xây một ngôi nhà thì nên thiết kế để sau này mở rộng để xây thêm phòng (cưới thêm vợ chặn hạn) chứ không phải mỗi lần sửa nhà là đập banh chành cái nhà cũ rồi xây lại một cái khác chỉ để có thêm một cái cầu ỉa hay phòng chơi xếp hình.

Giả sử bạn làm một chức năng tính thuế như sau:



Trong class này bạn có một thằng là **Calculate** nó nhận vào thu nhập của bạn, nước của bạn phải đóng thuế và thằng deduction tạm gọi là khấu trừ đi, deduction này phản ánh mức độ COCC của bạn.

Bạn implement thằng quỷ này như sau:

public decimal Calculate(decimal income, decimal deduction, string country)

{

decimal taxAmount = 0;

decimal taxableIncome = income - deduction;

switch (country)

{

case "India":

//calculation here

break;

case "USA":

//calculation here

break;

case "UK":

//calculation here

break;

}

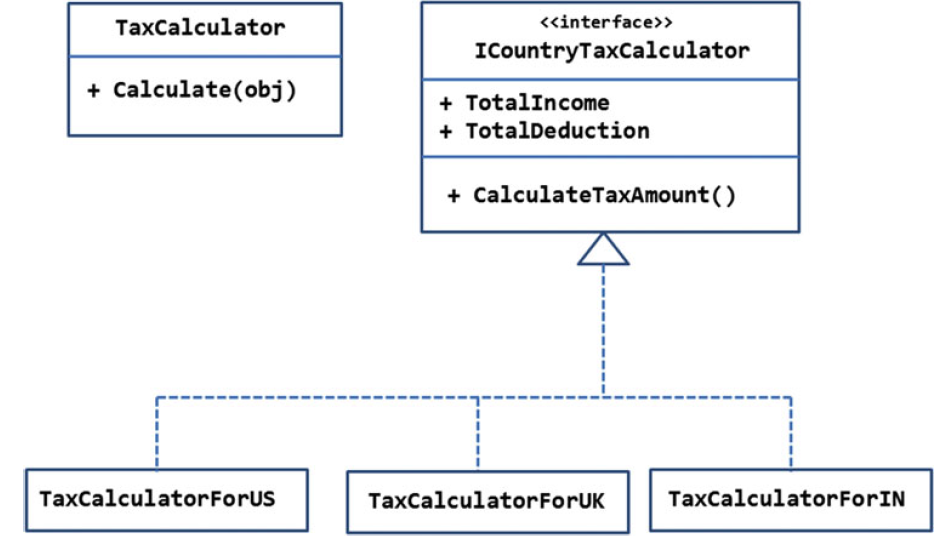
return taxAmount;

}

Bạn nhìn vào thiết kế bên trên và bạn vỗ mông và nghĩ răng **OK! It’s pretty awsome** rồi đấy.

Cơ mà một lần nữa cuộc đời khốn nạn lại tìm đến thằng DEV gà và lần này, user nói rằng, giờ cửa hàng của tao nó trở thành internation compnay và giờ tao muốn tính thuế cho tất cả tụi khách đến từ everywhere in the world. Và thế là thằng DEV gà ngày nào phải vào cái method mà nó đã implement phía trên để sửa lại khi có yêu cầu thêm một nước mới vào cái chỗ switch case và mỗi lần đó phải test đi test lại và nó ăn bugs ngập mồm luôn.

Bỗng một ngày chủ nhật đẹp trời khi nó đang fix bug cho việc thêm nước để tính thuế. Trong lúc nó đang khóc lóc ỷ ôi vì bug đầy mồm thì bụt xuất hiện và cho nó một sulution như sau:



Vừa lúc nó nhìn vào cái solution trên thì nó phải thốt lên “Dm, cái vẹo gì mà complicated quá vậy bug à nhàm bụt ?”.

Bụt bình tĩnh giải thích mà lòng nghĩ (dm thằng ngu).

Giờ mầy tạo một cái interface tính thuế cho tao. Thằng interface này nó có 2 thằng properties là **TotalIncome** và **TotalDeduction** và một method để tính là **CalculateTaxAmount**(). Và khi mầy muốn tính thuế một nước nào thì mấy tạo ra một class implement thằng interface đó và implement lại method **CalculateTaxAmount()** theo cách của nó. Và cuối cùng là mầy tạo ra class TaxCalculator và trong đó có method tính thuế tên là Calculate và nó nhận vào một kiểu cha của mấy thằng cần tính thuế là IcountryTaxCalculator, Việc của thằng method này là tao chỉ cần biết là tổng thuế của mày là bao nhiều, còn việc khai báo tính thuế là của tụi mày, sai sót là ở tù ráng chụi, việc của tao là cần biết cái cần biết mà thôi, còn cái khác tao méo quan tâm. Và giờ nếu có một thằng nước khác muốn báo thuế chỉ việc implement từ thằng interface và tự implement mấy thằng bên trong interface quy định là xong.

public interface ICountryTaxCalculator

{

decimal TotalIncome { get; set; }

decimal TotalDeduction { get; set; }

decimal CalculateTaxAmount();

}

public class TaxCalculatorForUS : ICountryTaxCalculator

{

public decimal TotalIncome { get; set; }

public decimal TotalDeduction { get; set; }

public decimal CalculateTaxAmount()

{

decimal taxableIncome = TotalIncome - TotalDeduction;

return taxableIncome \* 30 / 100;

}

}

public class TaxCalculatorForUK : ICountryTaxCalculator

{

public decimal TotalIncome { get; set; }

public decimal TotalDeduction { get; set; }

public decimal CalculateTaxAmount()

{

decimal taxableIncome = TotalIncome - TotalDeduction;

return taxableIncome \* 35 / 100;

}

}

public class TaxCalculatorForIN : ICountryTaxCalculator

{

public decimal TotalIncome { get; set; }

public decimal TotalDeduction { get; set; }

public decimal CalculateTaxAmount()

{

decimal taxableIncome = TotalIncome - TotalDeduction;

return taxableIncome \* 20 / 100;

}

}

public class TaxCalculator

{

public decimal Calculate(ICountryTaxCalculator obj)

{

decimal taxAmount = obj.CalculateTaxAmount();

//do something more if needed

return taxAmount;

}

}

1. **Liskov Substitution Principle (LSP)**

A type must be substitutable by its subtypes without altering the correctness of the application .

namespace Chapter1.Liskov

{

public class Class1

{

public string ProtertyClass1 { get; set; }

}

public class Class2:Class1

{

public string ProtertyClass2 { get; set; }

}

public class Class3 : Class1

{

public string ProtertyClass3 { get; set; }

}

class Program

{

static void Main(string[] args)

{

Class1 obj = null;

obj = new Class2();

obj = new Class3();

}

}

}

Đoạn chương trình trên hoàn toàn hợp lệ vì nó dựa trên tính kế thừa, nhưng nó sẽ lỗi ngay nếu class2 không còn kế thừa từ class1 hoặc class3 không kế thừa từ class1.

Ta cũng làm được tương như như vầy khi sử dụng Interface như sau:

namespace Chapter1.Liskov

{

public interface IInterface

{

void ShowInformation();

}

public class Class2 : IInterface

{

public string ProtertyClass2 { get; set; }

public void ShowInformation()

{

throw new System.NotImplementedException();

}

}

public class Class3 : IInterface

{

public string ProtertyClass3 { get; set; }

public void ShowInformation()

{

throw new System.NotImplementedException();

}

}

internal class Program

{

private static void Main(string[] args)

{

IInterface obj = null;

obj = new Class2();

obj = new Class3();

}

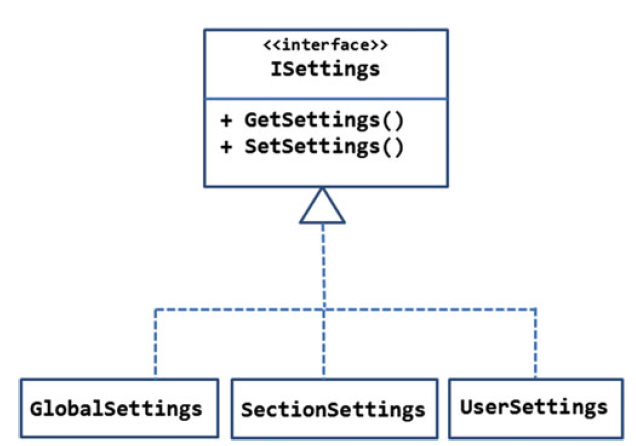
}

}

Từ hai ví dụ trên ta thấy rằng polymorphic behavior là một vũ khí lợi hại, nhưng vũ khí nào cũng có 2 mặt của nó cả, mặt trái của vũ khí là người sử dụng phải cẩn thận để không gây ra lỗi hoặc sự hoạt động không đúng của hệ thống.

Ta sẽ thảo luận về vấn đề này trong ví dụ sau đây:

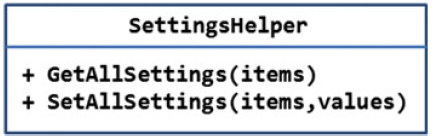
Bạn được sếp đẹp trai giao cho nhiệm vụ xây dựng một cái portal rất chi là hoành cm nó tráng, và vi hoành tráng nên sẽ có thêm tính năng tùy biến hệ thống. Tính năng này sẽ đáp ứng cho nhiều cấp độ sử dụng khác nhau như là global-level, section-level, user-specific level và bạn quất ngay cái thiết kế như sau:



Nhìn từ thiết kế trên ta thấy rằng

* Thằng ISettings có 2 methods là GetSettings() và SetSettings(), hai thằng này có nhiệm vụ là đọc setting từ DB và lưu lại setting bị thay đổi xuống DB.
* Thằng ISetting được implement từ 3 thằng là GlobalSettings, SectionSettings, UserSettings.
* Trách nhiệm của chú GlobalSettings là có mức cao nhất cho toàn hệ thống. (kiểu như giám đốc ấy).
* Còn chú SectionSettings thì chỉ cho từng section riêng biệt (có thể nghĩ đến là vị trí trưởng phòng chẳn hạn).
* Còn chú cuối cùng là UserSettings là cho setting cho mấy thằng user (kiểu như nhân viên ấy).

Giờ bạn tạo thêm một chú khác gọi là SettingsHelper để cover hết mấy thằng trên kia (kiểu như ông tổng ấy), thằng này hình dạng như sau:



Thằng này có 2 chú methods là GetAllSettings() và SetAllSettings().

Chú GetAllSettings() sẽ nhận một list những thằng mà implement từ thằng ISetting. Bên trong nó, nó sẽ gọi đi gọi lại thằng GetSettings() của từng thằng trong list truyền vào để lấy giá trị setting và cuối cùng nó sẽ trả về một cục bùi nhùi đó cho thằng gọi nó.

Trong khi đó chú SetAllSettings() cũng sẽ hành xử tương tự nhưng sẽ lưu lại các giá trị thay đổi truyền vào của mỗi thằng trong list items truyền vào bằng cách gọi method SetSettings().

// inside GetAllSettings()

foreach (ISettings item in items)

{

item.GetSettings();

}

// inside SetAllSettings()

foreach (ISettings item in items)

{

item.SetSettings(values);

}

So far, so good. Mọi thứ hoạt động ngon lành cành đào cho tới một ngày đẹp trời sếp gọi bạn lên và nói với bạn là giờ tao muốn cái portal này nó cũng cho thằng khách vãng lai (guest) sử dụng, mọi setting cho thằng này cũng giống như mấy thằng registered users nhưng chỉ một ngoại lệ là tụi này không được lưu bất cứ setting nào cho tụi nó cả (cho mầy sài là ngon rồi ở đó mà đòi hỏi) và tụi nó cũng sẽ có một vài setting mặc định cho tụi guest này.

Để giải quyết điều này, bạn liền tạo ra một class GuestSettings và cũng implement thằng Isettings. Vì yêu cầu là thằng guest chỉ có thể đọc mà không thể lưu dữ liệu do vậy bạn méo implement thằng SetSettings() như sau:

public void SetSettings(Dictionary<string, string> settings)

{

throw new NotImplementedException();

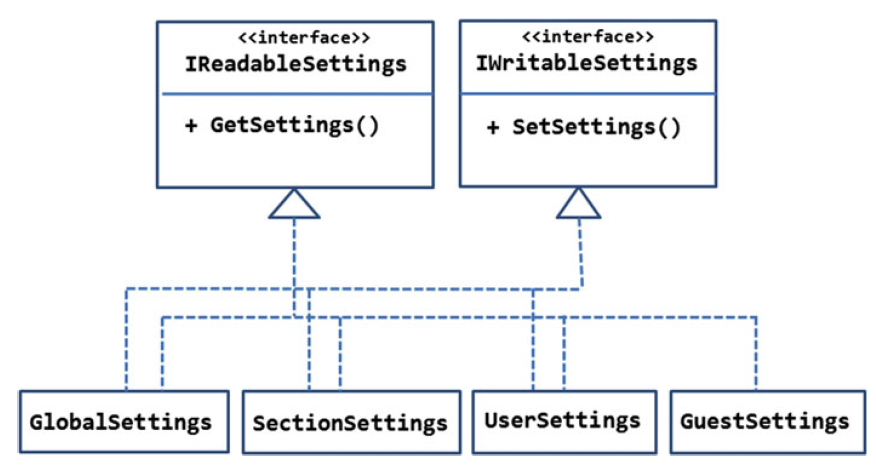
}

Và cuộc đời của bạn cũng sẽ bế tắc từ đây, since you implemented something like this :v

Mặc dù thằng GuestSettings implemented từ ISettings nhưng nó lại méo implement chú SetSettings(). Vậy khi vứt chú GuestSettings này vào chú method SetAllSettings() của anh đại (lớp SettingsHelper) thì chuyện gì sẽ xảy ra? Nó sẽ văng ra một lỗi vì việc của chú SetAllSettings() là gọi đến method SetSettings() của chú GuestSettings, mà chú này thì mình méo có implement theo yêu cầu của sếp. Vậy là xong, chuẩn bị CV là vừa.

Vấn đề bên trên là một ví dụ đơn giản cho việc vi phạm LSP (Liskov Substitution Principle).

Giải pháp là bạn phải tách chú ISettings ra thành 2 chú khác nhau là IReadableSettings và IWritableSettings như sau:



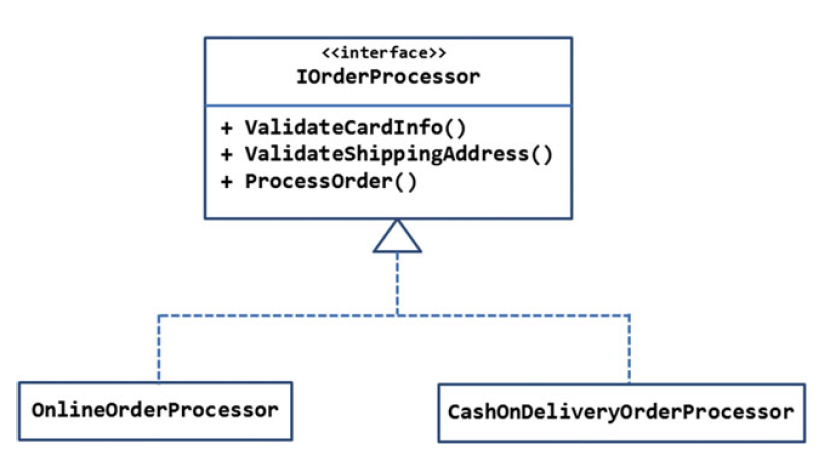
* Lúc này ta thấy rằng IReadableSettings nó sẽ chỉ có GetSettings() trong khi IWritableSettings thì có SetSettings().
* IReadableSettings được implement bởi tất cả các class trong khi IWritableSettings chỉ được implement bởi 3 classes là GlobalSettings ,SectionSettings , and UserSettings.
* SettingsHelper vẫn còn 2 thằng là GetAllSettings() and SetAllSettings(), nhưng lúc này chú GetAllSettings() sẽ nhận một list mấy chú IReadableSettings trong khi SetAllSettings() sẽ nhận một list mấy em IWritableSettings.

Vậy là xong rồi, mọi thứ following cái này thì điều đẹp đôi đường và không vi phạm thằng LSP (Liskov Substitution Principle) và cũng yên tâm cất CV đi :v .

1. **Interface Segregation Principle (ISP)**

***Clients of a class should not be forced to depend on those of its methods that they don’t use.***

Giả sử rằng bạn đang xây dựng một ứng dụng ecommerce mà cần đến hai chức năng là giỏ hàng và chức năng xử lý một chuỗi các order. Và thế là bạn sẽ thiết kế hệ thống như hình bên dưới.



Ban đầu bạn cho rằng chỉ chấp nhận thanh toán online mà thôi. Do vậy, bạn thiết kế một interface **IOrderProcessor** chỉ có 3 method là:

+ **ValidateCardInfo**() : Dùng để validate thông tin thẻ tín dụng như số thẻ và ngày hết hạn.

+ **ValidateShippingAddress**() : Thằng này để xác nhận địa chỉ mà sẽ giao hàng, để chắc rằng khúc vực mà khách đặt mua nằm trong tầm kiểm soát của bạn. (ví dụ như có một thằng nào đó vui tính ở bên đâu tận Triều Tiên vào trang của bạn mua cái s\*x toy có 25K, thế là bạn bán cái s\*x toy 25K mà phải mất phí ship hàng tận vài củ, easy money :v , vì vậy mà bạn phải cần method này để kiểm tra địa chỉ giao hàng).

+ **ProcessOrder**() : dùng để bắt đầu quá trình xử lý đơn hàng trong hệ thống của bạn.

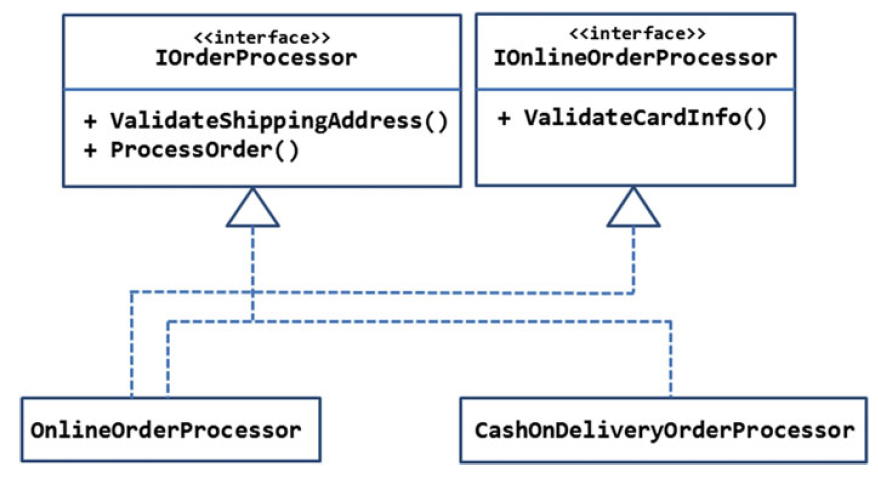
Ta thấy rằng class **OnlineOrderProcessor** implements thằng **IOrderProcessor** và nó hoạt động ngon lành cành đào như mục đính ban đầu.

Giả sử rằng một ngày đẹp trời Internet sập tiệm, thế là việc thanh toán online bằng thẻ tín dụng cũng tiêu tùng. Thế là ta phải tạo ra một chức năng dùng để thanh toán khi giao hàng (cash-on-delivery). Ban đầu, bạn nghĩ rằng, ôi đồ yêu này làm trong 3 nốt nhạc thôi! Bạn chỉ việc tạo một class **CashOnDeliveryProcessor** sau đó implements em **IOrderProcessor** vậy là xong.

Cơ mà có mùi gì đó méo ổn ở đây, cái dịch vụ thanh toán khi giao hàng này thì méo liên quan gì đến việc phải validate cái thẻ tính dụng làm quái gì, vì giờ cái việc này useless vl ra. Thế là bạn nảy ra sáng kiến là cái chú ValidateCardInfo() của **CashOnDeliveryOrderProcessor** chỉ việc vứt cái cục **NotImplementedException** ra mà thôi.

Đến lúc này, thì ứng dụng của bạn vẫn hoạt đồng bình thường chả có chuyện gì xảy ra cả, nhưng có một cái gì đó bất ổn ở đây! Giả sử rằng một ngày đẹp trời em thanh toán online bằng thẻ tính dụng đòi mở rộng ra thêm vài bước để validate. Theo thông lệ thì ông nội **CashOnDeliveryOrderProcessor** sẽ phải thêm mấy em methods mới để phục vụ cho nhu cầu của em thanh toán online bên trên (em **OnlineOrderProcessor** sẽ implement hết mớ này để sử dụng very happy). Cơ mà, chú **CashOnDeliveryOrderProcessor** thì chả có cần mấy cái vẹo này để làm gì nhưng vì chú ấy implement thằng ông nội **CashOnDeliveryOrderProcessor** nên cũng bị ép buộc là implement mớ method mà méo bao giờ dùng của em online payment kia. Nói một cách khác là em **CashOnDeliveryOrderProcessor** bị ép buộc là phải thay đổi để có mấy cái mà chả bao giờ cần đến (kiểu mấy ông chồng bị mấy bà vợ bắt mua cả đống thứ mà chả bao giờ dùng, cuộc sống như vậy có happy không?). Và với hành động này nó đã vi phạm đến **ISP**. Hơn nữa, với việc vứt cục nợ NotImplementedException từ những methods này, thì thằng **CashOnDeliveryOrderProcessor** cũng vi phạm nốt em **LSP**.

Giải pháp cho việc này như hình bên dưới.



Ta thấy rằng em **OnlineOrderProcessor** và em **CashOnDeliveryOrderProcessor** điều cần đến 2 chú là **ValidateShippingAddress**() và chú **ProcessOrder**() do vậy 2 class này phải implements ông nội **IOrderProcessor**.

Ở một khía cạnh khác em **OnlineOrderProcessor** cần phải có chức năng validate mấy cái thẻ nên phải có method **ValidateCardInfor**() do vậy em này sẽ implement **IOnlineOrderProcessor**. Và từ này về sau nếu muốn thêm functionality nào về online payment thì cứ việc nhét vào cái interface này. Còn nếu chức năng nào mà dùng chung cho cả hai loại phương thức thanh toán thì chỉ việc vứt vào **IOrderProcessor** là xong.

1. **Dependency Inversion Principle (DIP)**

***High-level classes should not dependent on low-level classes. Both of them should depend on abstractions.***

***Abstractions should not depend upon details. Details should depend upon abstractions.***

Thông thường thì bạn sẽ khởi tạo các dependencies (phụ thuộc) của một class ngay trong class đó và do vậy class của bạn càng càng càng bị trói buộc bởi các dependencies này. Bất kỳ sự thay đổi nào của một dependency cũng yêu cầu sự thay đổi theo ở trong class của bạn luôn. Mà cái nguyên nhân mấu chốt cho việc này là không ai khác là tự bạn trói bạn nghĩa là tự class đó tự tạo ra các dependency. Để giảm bớt sự phụ thuộc thì các dependencies nên được cung cấp từ một class bên ngoài.

DIP có 2 phần chính.

Thành phần đầu tiên nói về mối quan hệ giữa high-level và low-level classes.

* **High-level** class là những class làm những việc chuyên biệt chính yếu trong ứng dụng.
* **Low-level** class là những class mà phụ trách những công việc phụ trợ cho các class chính.

Ta sẽ xem sét xét ví dụ sau đây:

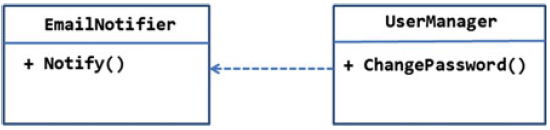
Ban đang xây dựng một thệ thống authentication và membership trên thiên địa và cần đến việc quản lý các em users (checkers nhà ta) và tất nhiên là chức năng thay đổi mật khẩu là phải có rồi. Khi mà việc thay đổi mật khẩu xảy ra thì tất nhiên sẽ có một message gởi tới user đó để báo “Ê account của mày có thay đổi đó nha”.

Trong ví dụ trên thì

Thì việc quản lý các em users là high-level class. Trong khí đó chức năng thay đổi mật khẩu là low-level class (cũng dễ hiểu thổi không có em đầu thì đếch có em tiếp theo nhé).

DIP phát biểu như sau: “Giai cấp bóc lột (high-level class) không nên phụ thuộc vào gia cấp bị bóc lột (low-level class) mà bọn chúng nên phụ thuộc vào abstractions”.

Thông thường thì anh high-level class sẽ tạo và sử dụng mấy em low-level class ngay chính bên trong nó theo cách đã làm day by day.



Nhìn vào hình bên tra ta thấy rằng em **UserManager** là high-level class em này có một method là **ChangePassword**(). Tuy nhiên, em lại bị trói buộc bởi anh **EmailNotifier** cho việc gởi message cho anh người yêu (cũng thường thôi, yêu nhau bị ngăn cách bởi bà chủ phòng trọ thôi mà). Chi tiết của sự trói buộc này được trình bày như sau:

public void ChangePassword(string username, string oldpwd, string newpwd)

{

EmailNotifier notifier = new EmailNotifier();

//change password here

notifier.Notify("Password was changed on " + DateTime.Now);

}

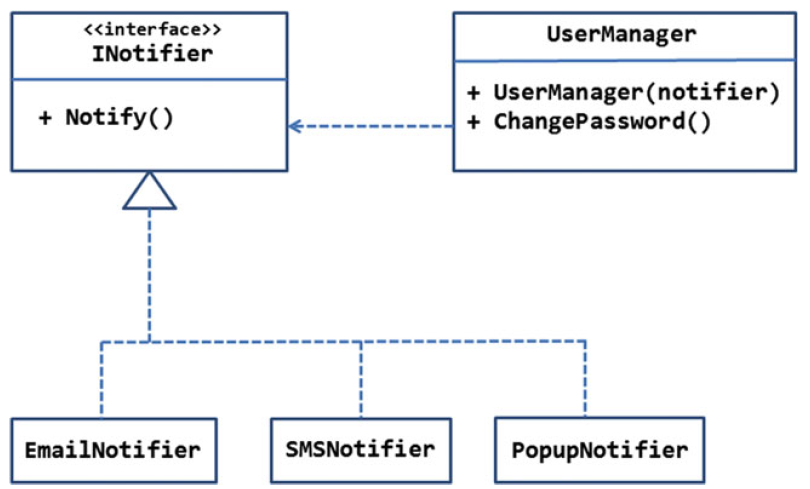
Như bạn thấy trong đoạn code phía trên thì **ChangePassword**() đi tạo ra một anh **EmailNotifier** rồi sau đó dùng em tạo ra đó gọi đến chú **Notify**() để thực hiện mục đích send message.

Bạn sẽ tự hỏi “thế có cái vấn đề với thiết kế này không? Bao năm nay anh vẫn sử dụng đó có ảnh hưởng đến hòa bình thế giới đâu?”.

Có đó, vấn đề ở đây là **UserManager** đã quá lệ thuộc vào em **EmailNotifier**, mỗi lần mà em **EmailNotifier** thay đổi một cái thì anh **UserManager** phải nhìn lại bản thân và phải thay đổi (kiểu anh chàng người yêu quá bánh bèo ấy). Hơn nữa, khi cần viết test cho anh **UserManager** thì em **EmailNotifier** phải được tạo ra không thì méo có test hàng gì hết nhé.

Hơn nữa, vào một ngày đẹp trời nào đó trong tương lai việc gởi nortification thay đổi cái thì anh **UserManager** tội nghiệp cũng phải lăn tăn xách dép thay đổi nốt theo. Chẳng hạn như bạn chia tay bé **EmailNotifier** vì gia đình không cho phép và đến với bé SMS thì bạn cũng phải thay đổi anh **UserManager** để đến với người mới (Chắc năng gởi nortification bằng SMS).

Để giải quyết vấn đề này thì DIP khuyên ta nên chỉ phụ thuộc vào abstraction. Thí chủ abstraction có thể là một based class hay một interface.



Nhìn vào quả hình bên trên thì ta sẽ thấy như sau:

* Anh bạn tội nghiệp **UserManager** không còn sử dụng trực tiếp em **EmailNotifier** nữa rồi, mà thay vào đó là một interface tên **INotifier**.
* Nhìn vào thí chủ **INotifier** là ông nội của em **EmailNotifier**. Do vậy, **EmailNotifier** phải implement mấy thứ của anh ông nội kia định nghĩa.
* Đến lúc này thì thằng constructor của **UserManager** sẽ nhận vào một instance của INotifier và method **ChangePassword**() sẽ sử dụng instance này để gọi đến chú **Notify**() của anh INotifier.
* Nếu giờ bạn muốn chia tay em **EmailNotifier** mà chuyển qua mấy em xinh đẹp khác như SMS hay popup thì cứ việc này cũng sẽ chẳng ảnh hưởng ăn nhậu gì tới anh thanh niên **UserManager** vì anh ta nói rằng anh chỉ làm việc với ông nội thôi còn mấy thằng cháo chít anh méo care nhé.

Phần thứ 2 của DIP nó rằng abstractions không nên phụ thuộc vào những chi tiết mà chi tiết nên phụ thuộc vào abstractions.

Điều này có thể hiểu như sau: bạn nên thiết kế **INotifier** (đây là em abstraction) thì nên nhìn vào những thứ mà chú **UserManager** cần đến chứ không phải những thứ mà em **EmailNotifier** (đây là em details) cần

**Creational Patterns: Singleton,Factory Method, and Prototype**

1. Overview of Creational Design Patterns.

Trong ứng dụng asp.net việc tạo mới một object là điều bạn sẽ làm rất thường xuyên và trông như sau:

Customer obj = new Customer();

Ở đoạn code phía trên bạn tạo một instance của class Customer bằng việc sử dụng từ khóa new. Trong hầu hết các trường hợp, bạn sẽ viết dòng code phía trên bên trong một method giống như ví dụ sau:

public class MyClass

{

public void DoWork()

{

Customer obj = new Customer();

obj.CustomerID = "ABCD";

obj.ProcessOrder();

}

}

Ta thấy rằng cách tạo đối tượng bên trong một method như trên là rất phổ biến. Nhưng ta sẽ thấy rằng sẽ có những trường hợp mà việc khởi tạo object bên trong một method là không nên dùng. Giả sử rằng bạn đang xây dựng một ứng dụng ASP.NET để hiển thị những biểu đồ cho người dùng. Bạn sẽ tạo một class để thực hiện việc hiển thị các biểu đồ này và chắc hẳn khi bạn sử dụng nó bạn sẽ thực hiện việc tạo mới nó theo cách phía trên.

Thời gian thấm thoát trôi đi, bạn quyết định ứng dụng web của bạn xây dựng trước kia giờ chia làm 2 versions là có phí và free. Với phiên bản trả phí bạn cung cấp cho người dùng những biểu đồ xịn hơn với phiên bản trả phí và vấn đề phát sinh từ đây. Giờ thì bạn bắt buộc phải vô thay đổi mấy dòng code chỗ khởi tạo mới cái class biểu đồ (charting). Và bạn nghĩ sẽ thật tuyệt biết bao nếu như có một class nào đó mà sẽ chụi trách nhiệm việc khởi tạo đối tượng cho tụi charting này và creational patterns ra đời để giúp bạn làm việc này.

Creational patterns sẽ tách biệt logic tạo mới đối tượng khỏi dòng code chính của ta. GoF đã nhóm lại và cho ta được 5 đứa là singleton, factory method, prototype, abstract factory, and builder.

* 1. Singleton

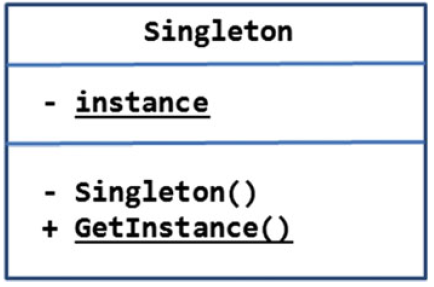
Thông thường bạn sử dụng từ khóa new để tạo mới đối tượng, bạn new đối tượng 5 lần thì bạn cũng nhận được 5 đứa khác nhau. Vậy bạn sẽ làm sao nếu bạn chỉ muốn chỉ duy nhất, độc nhất một đứa được tạo thôi? Vâng! đã có anh hùng singleton ra giúp bạn.

Singleton phát biểu như sau:

**The purpose of the singleton pattern is to ensure that only one instance of a class is created.**

Những class mà implement singleton pattern không những đảm bảo là chỉ duy nhất một thực thể được tạo mà còn cung cấp để truy cập thực thể đó. Thực thể được tạo ra ở lần request đầu tiên.

1. Design and Explanation



Bên trên ta thấy class Singleton implements singleton pattern. Ta thấy ở class này khai báo một biến (**instance**) kiểu static có mức truy cập là private và vì vậy **instance** không thể nào được truy cập từ tiếp từ bên ngoài.

Ta cung thấy được rằng phương thức khởi tạo của lớp cũng có mức truy cập là private luôn và do vậy bạn cũng không thể tạo mới thực thể bằng cách sử dụng từ khóa **new**. Tuy nhiên bạn lại có thể làm được bên trong nội bộ class.

GetInstance() là một static method và có mức truy cập public. Phương thức này sẽ trả về thực thể (instance) cho đứa mà gọi đến nó, nó đóng vai trò là một access point cho thực thể (instance). Nói một cách dễ hiểu thì chú method GetInstance() sẽ trả về một thực thể nếu nó đã tồn tại và tạo mới nếu nó chưa tồn tại (nếu đó là lần gọi đầu tiên đến nó).

Thiết kế theo hình vẽ bên trên khi ta cho rằng class Singleton sẽ không được thừa kế bởi các class nào khác, nếu nó thiết kế theo hướng mà được kế thừa thì ta nên để constructor (phương thức khởi tạo) nên là protected hơn là private.

1. Example

Giả sử rằng bạn sẽ xây dựng một ứng dụng một web blog. Bạn sẽ thấy rằng những thứ như địa chỉ email liên lạc, error-logging status, theme, và title của blog sẽ được sử dụng trong toàn bộ tất cả các module như layout pages, views, components. Thay vì ta cứ phải tạo đối tượng ở từng module này thì ta có thể chỉ cần tạo một lần duy nhất và sử dụng nó ở toàn bộ các module.

Lược đồ UML cho thiết kế này như sau:



[Table("WebsiteMetadata")]

public class WebsiteMetadata

{

[DatabaseGenerated(DatabaseGeneratedOption.Identity)]

public int Id { get; set; }

[Required]

[StringLength(100)]

public string Title { get; set; }

[Required]

[StringLength(40)]

public string DefaultTheme { get; set; }

[Required]

[StringLength(50)]

public string AdminEmail { get; set; }

[Required]

public bool LogErrors { get; set; }

private static WebsiteMetadata instance;

private WebsiteMetadata()

{

}

public static WebsiteMetadata GetInstance()

{

if (instance == null)

{

using (AppDbContext db = new AppDbContext())

{

if (db.Metadata.Count() == 0)

{

db.Metadata.Add(new WebsiteMetadata()

{

Title = "My Application",

AdminEmail = "admin@localhost",

DefaultTheme = "Summer",

LogErrors = true

});

db.SaveChanges();

}

instance = db.Metadata.SingleOrDefault();

}

}

return instance;

}

}

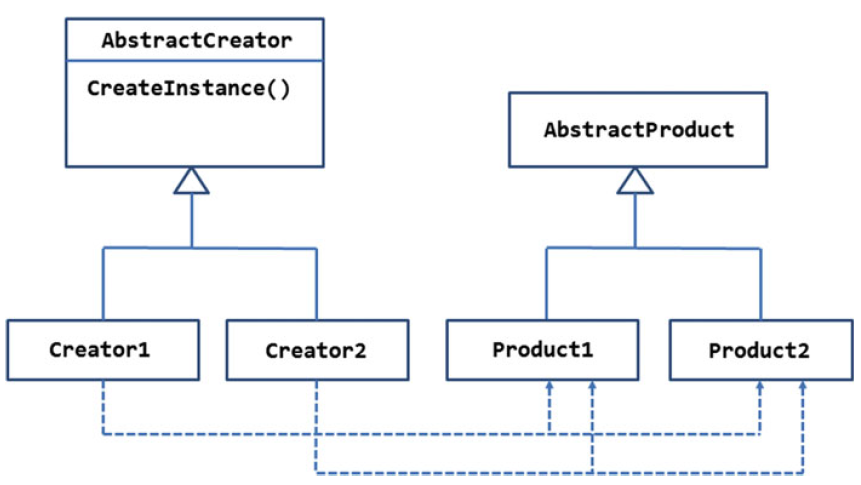
Nhìn vào đoạn code bên trên ta thấy rằng:  
- Static method GetInstance() là một access point của **instance** của thực thể. Đầu tiên nó kiểm tra **instance** đã được được gán giá trị nào chưa, khi ta run ứng dụng lần đầu tiên thì **instance** sẽ mang giá trị là null. Ở ngữ cảnh này nếu **instance** là null, chương trình sẽ chạy xuống database và lấy thông tin metadata cho nó. Ta cần chú ý một chút nếu table **WebsiteMetadata** mà chưa có dữ liệu nào chương trình sẽ thực hiện thêm những giá trị mặc định cho table này, còn nếu đã có dữ liệu thì nó chỉ lấy lên cho biến instance.

Bạn thấy rằng đoạn code phía trên không tạo mới thực thể của WebsiteMetadata bằng việc sử dụng từ khóa new bởi vì Entity framework đã làm điều này cho ta.

* 1. Factory Method

Factory method pattern phát biểu như sau**: The factory method pattern defines a way to create an object. And the subclasses decide which class to instantiate.**

1. Design and Explanation



Dựa vào lực đồ bên trên ta sẽ thấy rằng  
- Một lớp trừu tượng có tên là AbstractCreator chứa một method có tên là CreateInstance(). lớp này chính là một template của factory method.

- AbstractCreator được kế thừa bởi 2 class con là Creator1 và Creator2 và do vậy chúng phải implement lại method CreateInstance().

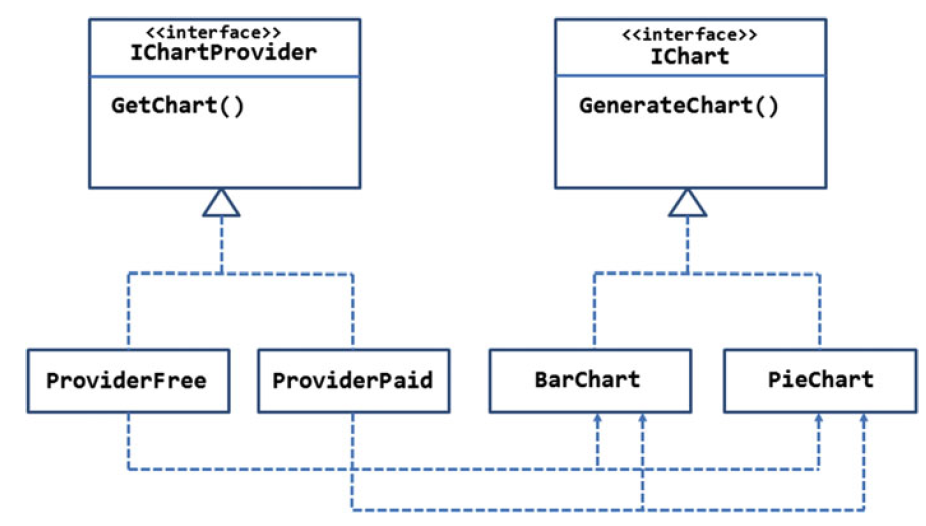
- Khi ta implement method CreateInstance() thì tại lại cần phải quyết định chọn khởi tạo Product1 hay là Product2. Product1 và Product2 lại nhà những class con của class AbstractProduct.

- Sự lựa chọn là sẽ intentiate Product1 hay Product2 thì lại phụ thuộc vào các logic của application.

- Về phía client, client có thể là chọn Creator1 hay Creator2 và khi gọi đến CreateInstance() thì lại có thể lựa chọn tiếp là chọn Product1 hay Product2 (điều này phụ thuộc vào logic của việc khởi tạo) và do vậy client có được cái họ muốn.

**b. Example**

Ngữ cảnh: Giả sử rằng bạn sẽ xây dựng một ứng dụng mà có các thành phần bao gồm: công nhân, thời gian của họ trên một công việc. Và bạn xây dựng một chức năng là hiển thị số giờ làm việc của mỗi ngày của một tuần lên một biểu đồ. Biểu đồ thì có 2 dạng là dạng cột hay dạng tròn. Và người dùng sẽ lựa chọn dạng biểu đồ nào để show thông tin mà họ muốn xem. Sau khi xem sét tất cả các requirements bạn đi đến một thiết kế như bên dưới



Nhìn vào thiết kế bên trên bạn thấy rằng:

Interface IChart được định nghĩa một method là GenerateChart(). Cái method này được dung để tạo ra một biểu đồ dựa vào tọa độ X và Y. Interface IChart được implemtn bởi 2 class con là BarChart và PieChart.

Trong khi đó, thằng IChartProvider định nghĩa một factory method (một nhà máy method à? Không nên dịch vậy nghe ghê vl), factory method này tên là GetChart(), khi được gọi đến thì method này nó sẽ trả về thằng Barchart hoặc PieChart dựa vào ý muốn của người dung.

Thằng IChartProvider được implement bởi 2 class con là ProviderFree và ProviderPaid. Và những thằng này phải implement method GetChart() của thằng cha.

Thật ra thì bạn không phải lúc nào cũng phải có nhiều thằng implement GetChart() như bên trên. Ta có thể chỉ cần một “creator” để implement GetChart() cho việc trả về là BarChart hay PieChart. Tuy nhiên, giả sử rặng bạn có 2 loại user account – loại miễn phí và loại phải trả phí. Nếu user là đối tượng miễn phí thì bạn phải dùng ProviderFree để tạo ra các đối tượng chart cho họ, trong khi đó nếu user là đối tượng trả phí thì ta cần dùng ProviderPaid để tạo ra các đối tượng chart cho họ. Tại sao phải làm vậy ư? Vì đối với đối tượng trả phí thì có thể bạn cho họ sài nhiều tín năng hơn để hiển thị dữ liệu chẳng hạn hay nhiều hiệu ứng màu mè canh lá hẹ chẳng hạn?

Bây giờ ta sẽ cùng xem xét một demo bằng ASP.NET cho factory mthod này bên dưới.

* 1. Prototype