سلسلة الأساسيات في تصميم البرمجيات

البرمجة باستخدام ال Interface

لبناء أساس متين في تصميم البرمجيات

وجدي عصام عبد الرحيم

جدول المحتويات

مقدمة الكتاب

كلنا نعرف وجود ال interface في لغات البرمجة مثل Java، C# ولكن لا نعرف متى نستخدمه بطريقة فعالة Effectively ومتى يلعب دور مهم في تصميم المشروع Software Design.

أغلب البرمجيات المبنية جيداً تعتمد على وجود ال Interface فيها، فال Interface يعتبر العمود الأساسي في الكثير من العادات الجيدة في البرمجة Modern Techniques مثل ال Testability، وال Dependency Injection وكثير من ال Design Patterns، وحتى في أساسيات التصميم التي تعرف ب SOLID Design Principles. لذلك فهم ال Interfaces وأين تستخدمه بالشكل الصحيح يعتبر خطوة أولى للمواضيع الأكثر تقدماً في عملية تصميم البرمجيات بالشكل الصحيح وهو الخطوة لكي تصبح مبرمج أكثر دراية بطرق التصميم ومعماريات الأنظمة.

في هذا الكتاب سوف نبدأ الحديث عن ال Interface بدءاً من السؤال الشائع ألا وهو الفرق بينه وال Abstract Class ومن ثم سنتدرج بالحديث عن كيف يساعد ال interface في جعل الكود أكثر قابلية للصيانة Maintainability، وكيف يكون الكود قابل للتطوير بسهولة لإضافة المزيد من الخصائص بدون الحاجة لتغيير الكثير من الكود Extensibility، وكيف تحمل الImplementations المختلفة وقت التشغيل Dynamic Loading أو ما يعرف ب Late Binding.

بعد ذلك سوف نأخذ مثال يحتوي مجموعه من المفاهيم في تصميم البرمجيات، ابتدئاً بفكرة تقسيم الاهتمامات Seperating of Concerns ومروراً بالطبقات في المشروع Larying وكيف يمكن بناء تلك الطبقات مع التفاعل بينهما.

لمن هذا الكتاب؟

هذا الكتاب موجه للمطورين ومهندسي الأنظمة والمبتدئين في هندسة البرمجيات، سواءً كنت جديداً في تصميم البرمجيات أو لديك عدة سنوات خبرة فسوف تستفيد من هذا الكتاب.

المستوى العام للكتيب هو **مبتدئ** ولا يتطلب أي معرفة متقدمة، وطالما أنك تعرف اساسيات البرمجة وال Object Oriented فسوف تستطيع فهمه واستيعاب ما فيه بلا مشاكل.

الأمثلة البرمجية ستكون باستخدام C# ولكن مبرمجي الجافا والسي++ وأي لغة برمجة تدعم الاسلوب الكائني First Class Object Oriented Language فبإمكانهم الاستفادة من الكتاب. حيث المفاهيم عامة وتطبق على أي لغة وتم الابتعاد عن استخدام المزايا الخاصة في اللغة بقدر الإمكان وجعل الكود أكثر عمومية حتى يستفيد مبرمجي اللغات الأخرى. ويمكنك إذا أردت أن تتبع أمثلة الكتاب أن تحمل نسخة Visual Studio Express 2015 أو 2013 وتطبق ما هو موجود (وهي مجانية بالمناسبة).

لم نقم بترجمة المصطلحات وحاولنا بقدر الإمكان كتابة المصطلح فقط (كما في عنوان الكتاب مثلاً ال Interface) حتى يكون أسهل في الفهم والاستيعاب.

سيتم عرض كافة أكواد الأمثلة التي يتم طرحها، وهي مرفقة مع الكتاب إذا أردت مشاهدتها، وفي حال لم تجدها مع الكتاب فيمكنك تحميل كل الأمثلة من خلال هذا الرابط

ملخص محتوى الكتاب

- **الفصل الأول "مراجعة سريعة"**: وسوف نأخذ مثال ويتم برمجته بعدة طرق بدءاً من الطريقة التقليدية Procedures ثم الانتقال لل Classes واستخدام ال Abstraction بأنواعها الثلاثة سواء الوراثة من الكلاس العادي Concrete class أو الوراثة من Abstract Class أو استخدام الinterface ، ومن ثم يتم النظر لتلك الحلول ومشاهدة الانسب للمشكلة، واخيراً يتم طرح الفروق بين ال Abstract class وال Interface ومتى تستخدم كل منهم.

- **الفصل الثاني: نظرة حول ال Interface**: واحدة من أهم الأسباب هو أنه يساعد في جعل الكود أكثر قابلية للصيانة Maintainableونعنى بها الكود المرن في التغيير وليس الذي يحتاج تغييره كاملاً بسبب تغيير بسيط في المتطلبات. وسوف نرى كيف يمكن أن تفيدنا ال Interface في ذلك وبالتالي يضمن هذا الأمر أن الكود يعمل للمستقبل Future Proof Code.

- **الفصل الثالث: كيف تستخدم ال Interface بكفاءة** سوف نرى في هذا الفصل كيف يمكن أن تستخدم ال Interface في جعل الكود قابل للتطوير بسهولة، حيث نريد انشاء الكود ال Extensible أي القابل للتعديل بسرعه بعد تغيير المتطلبات، وسيتم شرح بعض الأمور في سياق الموضوع مثلاً ال Repository Pattern وال CRUDsوكيف يمكن كتابة Static Factory.

- **الفصل الرابع: ال Dynamic Factory** سوف يتم تناول كيف يمكن تحميل ال Implementation وقت التشغيل وذلك ما يعرف ب ال Late Binding، وشرح الفروقات بين ال Compile Time Factory وال Dynamic Factory، وكيف يمكن جعل التطبيق يحمل أي Implementation جديدة من خلال تحديدها في ملف خارجي Configuration File بدون تغيير الكود أو اعادة ترجمته.

- **الفصل الخامس: معمارية الأنظمة وال Layering** سوف يتم تناول مثال مبسط عن ال Interface يحتوي أيضاً عديد من الأفكار في تصميم البرمجيات، بدئاً من فكرة تقسيم الاهتمامات إلى Layers ومروراً بال Business Layer وال Service Layer وال Data Access Layer واخيراً واجهة العرض للمستخدم. وسوف يتم المرور بعدة Patterns في التصميم خصوصاً في ال Business Layer مثلاً Null Object Pattern، Strategy Pattern وفي الواجهة سوف استخدام ال Model View Presenter.

- **الفصل السادس: خاتمة** ويتم وضع خلاصة لموضوع هذا الكتاب وكيف تستفيد من ال Interface بطريقة صحيحة.

- **الملحق 1: استخدام المكتبات Dependencies في سي#:** سوف يتم شرح انواع ال Dependencies وكيف يمكن أن تنشئ مشروع مكتبة Class Library وتستخدمها في مشروعك. وكيف تستطيع معرفة كل ال Assemblies التي تم تحميلها اثناء تشغيل البرنامج.

- **الملحق 2: نقاط متفرقة حول ال Interface في السي#:** سوف يتم تناول بعض الأمور المتعلقة بال Interface الخاصة في لغة سي# وكيف تتم الوراثة في ال Interface.

الأساسيات في تصميم البرمجيات

هذا الكتاب يعد أول جزء في هذه السلسلة "**الأساسيات في تصميم البرمجيات**"، والتي نهدف منها أن تكون شاملة في مجال تصميم البرمجيات بالطرق الحديثة، وأن تعد مهندس برمجيات Software Engineer قادر على تصميم برمجيات بطريقة سليمة وبأساسيات ومعرفة صحيحة، وفي نفس الوقت نزيد من كفاءة المطورين الحاليين بحيث يستطيعوا اصلاح البنية التحتية في المشاريع التي يعملوا بها.

الأجزاء التي نريد أن نخرجها هي:

- **الجزء (1) مقدمة في تصميم البرمجيات**
  - وهي شرح المفاهيم والمصطلحات المهمة في تصميم البرمجيات:
    - Software Design, Software Architecture, Software Development
    - Flexibility, Security, Reliability, scalability, Readability, Maintainability
    - Loosely Coupled, Highly Coupled, Highly Cohesive code, tightly coupled
    - Code Smell, Technical Debt, Anti-Patterns, Refactoring
  - شرح بعض الفلسفيات في التصميم
    - KISS: Keep it Simple Stupid
    - DRY: Do not repeat yourself
    - Tell, Don't Ask
    - YAGNI: you Ain't goanna need it
    - SOC: Separation of concerns
- **الجزء (2) عن ال Interface Based Programming**
  - وهو الذي بين يديك الآن
- **الجزء (3) عن أفكار Report Martin المعروف ب Uncle Bob والتي تعرف ب SOLID**
  - SRP: Single Responsibility Principle
  - OCP: open closed principle
  - LSP: Liskov substation principle
  - ISP: Interface Segregation Principle
  - DIP: Dependency Inversion principle
- **الجزء (4) عن Dependency Injection & IoC & Containers**
  - Dependency Injection
  - Dependency Inversion Principle & Inversion of Control
  - ما هي تلك المصطلحات والفرق بينهم والفائدة من كل مفهوم منهم وأين يستخدم
- **الجزء (5) عن ال Software Architecture Patterns**
  - شرح أنواع ال Design Patterns والأنماط التي طروحها ال Gang of Four
    - مع أمثلة عملية على الأنواع
  - وقفة عن ال Enterprise Design Patterns التي وضعها Martin Flower
    - شرح للطبقات Layers ووصف ما يجري بداخل كل طبقة وال Patterns المستخدمة
      - طبقة ال Data Access Layer
        - Repository & DAO

        - Lazy Loading & ORM & Unit of Work
    - طبقة ال Business Layer
        - Transaction Script & Active Record
        - Anemic Domain Model & Domain Model
    - طبقة ال Service Layer
        - Messaging Patterns & SOA
    - طبقة ال Presentation
        - MVP & MVC
        - الحديث حول Command/Query Responsibility Segregation واختصاراً CQRS
- **الجزء (6) الحديث عن مواضيع متفرقة في تصميم البرمجيات:**
    - الاختبارات بأنواعها المختلفة
        - Test Driven Design - TDD & Unit Testing
    - Domain Driven Design - DDD
    - Behavior Driven Design – BDD
    - Aspect Oriented Programming – AOP
- **الجزء (7) الأدوات في هندسة البرمجيات:**
    - ادوات ال Unit Testing & Continuous Integration & Documentations
    - ادارة المشروع بين الفريق Source Control Workflow
    - أدوات اكتشاف الأخطاء في الكود

هذه النقاط ليست كل شيء، لا في العناوين ولا في ترتيب الأجزاء وانما هي الفكرة العامة والخطوط العريضة للمواضيع المراد طرحها، ونعلم أن الطريق طويلة، ولكن الحمد لله فتم انجاز أول جزء منه وسوف يتبعها البقية أن شاء الله، وآمل أن يشاركوا في العمل المهتمين في مجال هندسة البرمجيات والمشاركة في الكتابة معي لإتمامها ومراجعتها حتى تكون هذه السلسلة المرجع الأول لأي مهندس برمجيات يريد البدء بالمجال، ويمكنكم التواصل معي بهذا الخصوص عبر بريدي الموضح أدناه أو اضافة الأسئلة والنقاش في أي موضوع في السلسلة أو هذا الكتاب وذلك من خلال مجتمع "الأساسيات في تصميم البرمجيات" على موقع حسوب.

ومرحباً بك في هذا الجزء، وأتمنى أن أكون قد وفقت في ايصال مادة علمية/عملية مفيدة للقارئ.

عن الكاتب

مهندس برمجيات يعمل في المجال منذ أكثر من 7 سنوات، طور العديد من الحلول والمشاريع بلغات ومنصات مختلفة. يعمل حالياً مدير قسم تطوير الحلول في مركز التميز لأمن المعلومات.

للتواصل مع الكاتب:

- البريد الالكتروني: wajdyessam@hotmail.com
- مقالات الكاتب: موقع انفورماتيك، مؤلفات للكاتب: موقع مؤلفات
- للأسئلة العامة والنقاشات: صفحة الكتاب على حسوب الأساسيات في تصميم البرمجيات
- صفحات انفورماتيك على Facebook أو Twitter

الثلاثاء 2016/Jan/05

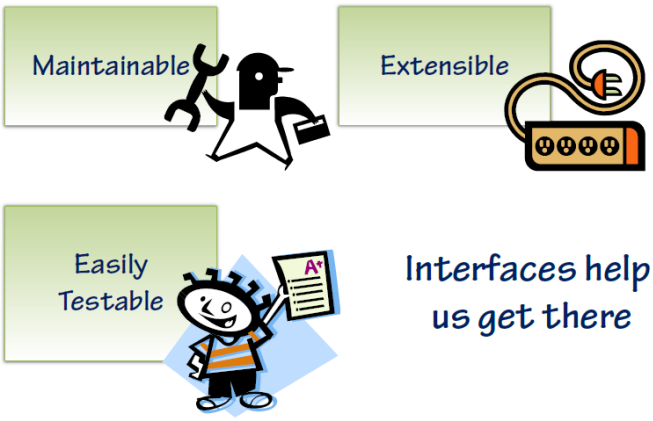
الفصل الأول: مراجعة سريعة

ما هو ال Interface

ببساطة هو Abstract type لا يوجد فيه أي Implementation أو Data فقط تصاريح للدوال Declarations.
وأفضل طريقة للتفكير في ال interface هو أنه عقد contract وله أعضاء members مثل ال methods وال Properties (ال Properties هي في السي# ويمكن اعتبار وجودها في الInterface على أنه بمثابة تصريح لمتغير)، وعندما يقوم أي كلاس أخر بعمل Implements لذلك العقد فيجب عليه أن يقوم بتطبيقه بالكامل ويعرف كل ما هو موجود في ذلك ال interface

لاحظ أن كل الأعضاء سوف تكون public تلقائياً (بفكرة العقد فلا يمكن أن يكون هناك عقد غير معروف بنوده، وأن جميع بنوده يجب أن تكون معروفة public).

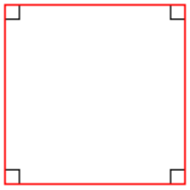
استخدام ال Interface وال Abstract Class يضيفان طبقة من ال Abstraction في الكود، والفروقات التي بينهم هي التي تحدد أي Abstraction سوف تفضلها للمشكلة. وكما سنرى خلال فصول هذا الكتاب سوف تعرف الفروقات ومتى تختار منهم (هذا الفصل)، وبعدها سوف نرى كيف تسهل لك ال Interface قابلية الصيانة (الفصل الثاني)، وقابلية التطوير (الفصل الثالث)، وسهولة الاختبارات (ال Unit Testing خارج محتوى الكتاب).



ال Interface يساهم في الثلاثة خصائص أعلاه

المثال الأول:

سوف نبدأ بمثال عملي بسيط نوضح فيه المفاهيم والأساسيات بين الInterface مع ال Abstract class والكلاس العادي Concrete (سوف نطلق على أي class عادي ليس بAbstract بالاسم Concrete class)، وليكن لدينا برنامج لحساب مساحة ومحيط المربع من خلال ادخال طول الضلع وتقوم بإظهار المحيط والمساحة.



مربع متساوي الاضلاع

بالطبع هذا الموضوع ليس في الرياضيات (واطمئن لن نكمل كثيراً في هذا المثال) وانما هو مجرد بداية توضيحية، وقوانين حساب المحيط والمساحة للمربع كالتالي:

- **مساحة المربع** = طول الضلع في نفسه
- **محيط المربع** = 4× طول الضلع

بمعنى أن المدخل سوف يكون طول الضلع، والمطلوب منك كتابة برنامج يقرأ المدخل ومن ثم يطبع المساحة والمحيط.

بالطبع أي مبرمج سوف يبدأ من الmain ويكتب كود بسيط يطبق تلك المعادلات، كما في الشكل:

```
1.  static void Main(string[] args)
2.  {
3.      // Square
4.      int numberOfSides = 4;
5.
6.      // Calculate Area
7.      Console.Write("Enter Side Length: ");
8.      int sideLength = int.Parse(Console.ReadLine());
9.      double area = sideLength * sideLength;
10.     Console.WriteLine("Area: " + area);
11.
12.     // Calculate Perimeter
13.     double permieter = numberOfSides * sideLength;
14.     Console.WriteLine("Permieter: " + permieter);
15.
16.     Console.ReadKey();
17. }
```

تعديل 1: الآن لو طلبنا من اضافة خاصية جديدة لهذا البرنامج وهي حساب محيط ومساحة المثلث متساوي الأضلاع ايضاً من خلال معرفه طول أحد الأضلاع، من خلال المعادلة التالية:

- **مساحة المثلث** = $\frac{a^2\sqrt{3}}{4}$ حيث a هي طول الضلع.
- **محيط المثلث** = 3 * طول الضلع

قد تكتب نفس الكود التالي:

```
1.  static void Main(string[] args)
2.  {
3.      // Triangle
4.      int numberOfSides = 3;
5.
6.      // Calculate Area
7.      Console.Write("Enter Side Length: ");
8.      int sideLength = int.Parse(Console.ReadLine());
9.      double area = sideLength * sideLength * Math.Sqrt(3) / 4;
10.     Console.WriteLine("Area: " + area);
11.
12.     // Calculate Perimeter
13.     double permieter = numberOfSides * sideLength;
14.     Console.WriteLine("Permieter: " + permieter);
15.
16.     Console.ReadKey();
17. }
```

إلى هنا الأمر جميل وأنت تعمل بطريقة اجرائية Procedure أو لنقل بأن برنامج بسيط للغاية ولا يخرج من كونه Toys Example.

لاحظ أن الفرق بين الكودين (الأول والثاني) وهو يكون فقط في الأسطر (4 و9) وبقية الكود هو كود متكرر. **ودائماً** في حال هناك تكرار في الكود فعليك أن تعرف أن هناك **تصميم غير جيد**، وأن الكود قد يقع في مشاكل عديدة مع أي تغيير أو اضافة تحدث.

والتغييرات دائماً ما تحصل، قد تتساءل كيف سيحصل التغيير في هذا الكود البسيط؟ الجواب كما حدث في تعديل 1 حيث تم اضافة المثلث فقد يمكن اضافة اشكال اخرى، أو ايضاً يمكن أن يحدث في عدة أماكن أخرى، مثلاً طريقة الإدخال، فبدلاً من أنك تدخل طول الضلع من لوحة المفاتيح، سوف يطلب منك أن تقرأ طول الضلع من ملف خارجي به وصف لأشكال هندسية مثل مربع ومستطيل ومثلث ونريد أن تقوم بحساب المساحة والمحيط لكل شكل موجود في الملف.

تعديل 2: الآن طلب أن يتم قراءة معلومات الأشكال من ملف خارجي (سواء كان XML أو أي ملف بأي صيغة) المهم به تعريف تلك الاشكال وعليك أن تقرأها جميعاً من ذلك الملف ومن ثم تحسب المساحة والمحيط.

هذه اللحظة سوف تحتاج لأن تتعامل مع المفهوم Concept بدلاً من أن تتعامل مع متغيرات مثلاً numberOfSide وarea فالأفضل أن تتعامل مع المفهوم نفسه (الشكل الهندسي) سواء كان Triangle أو Square وبالتالي تستطيع أن ترجع من دالة القراءة من الملف مجموعه من الأشكال Shapes، وهذه أولى فوائد ال Object Oriented حيث تستطيع تمثيل المفاهيم بطريقة أفضل بدلاً من التعامل مع متغيرات.

لكن المشكلة أنك الأن سوف تحتاج لأن ترجع جميع الأشكال في الملف مع بعضها، والا سوف تقوم بعمل دالة لإرجاع المثلثات، ودالة لإرجاع المربعات والخ، ولقد تم اخبارك أن هذه الاشكال سوف يتم عمل مجموعه من العمليات عليها فيما بعد مثلاً عرضها على الشاشة أو تطبيق مجموعه من الحسابات بها. لذلك يجب أن تستخدم مفهوم موحد Abstraction يشمل هذين الشكلين وبقية الأشكال في المجموعة.

لتطبيق مفهوم الشكل Abstraction قد تقوم بأكثر من طريقة:

1. عمل Concrete class (كلاس عادي) يمثل الشكل مثلاً نسميه Regular Polygon (يمكنك تسميته بأي اسم) ومن ثم كلاس المربع والمثلث يرثان Inheriting من ذلك الكلاس.

2. عمل Abstract class يمثل الشكل Regular Polygon ومن ثم والمربع والمثلث يرثان منه
3. عمل Interface اسمه Regular Polygon ويقوم المربع والمثلث باستخدامه Implement

سوف نتناول هذه الحلول ولنرى ماهي الفوائد والعيوب في كل حل منهم.

تطبيق ال Abstraction باستخدام الكلاس العادي Concrete Class

لاحظ الكود التالي يقوم بعمل class يمثل الشكل وفيه عدد الأضلاع وطول الضلع، بالإضافة إلى دالة حساب المحيط والتي هي مشتركة بين المثلث والمربع، اما دالة حساب المساحة فطالما الشكل غير محدد فسوف يتم عمل throw لل Exception في حالة تم استدعاء هذه الدالة.

```
1.  public class ConcreteRegularPolygon
2.  {
3.      public int NumberOfSides { get; set; }
4.      public int SideLength { get; set; }
5.
6.      public ConcreteRegularPolygon(int sides, int length)
7.      {
8.          NumberOfSides = sides;
9.          SideLength = length;
10.     }
11.
12.     public double GetPerimeter()
13.     {
14.         return NumberOfSides * SideLength;
15.     }
16.
17.     public virtual double GetArea()
18.     {
19.         throw new NotImplementedException();
20.     }
21. }
```

الآن نقوم بتعريف الكلاس Square ولاحظ أنه يرث من الشكل ويقوم بعمل override للدالة المساحة.

```
1.  public class Square : ConcreteRegularPolygon
2.  {
3.      public Square(int length) :
4.              base(4, length)
5.      {
6.      }
7.
8.      public override double GetArea()
9.      {
10.         return SideLength * SideLength;
11.     }
12. }
```

نفس الأمر مع الكلاس الTriangle :

```
1.  public class Triangle : ConcreteRegularPolygon
2.  {
3.      public Triangle(int length)
4.              : base(3, length)
5.      {
6.      }
```

```
7.
8.      public override double GetArea()
9.      {
10.         return SideLength * SideLength * Math.Sqrt(3) / 4;
11.     }
12. }
```

لننظر الآن إلى استخدام تلك الكلاسات في الدالة الرئيسية، ولتجربة كلاس المربع Square:

```
1.  static void Main(string[] args)
2.  {
3.      Square square = new Square(5);
4.      Console.WriteLine("Area: " + square.GetArea());
5.      Console.WriteLine("Permieter: " + square.GetPerimeter());
6.
7.      Console.ReadKey();
8.  }
```

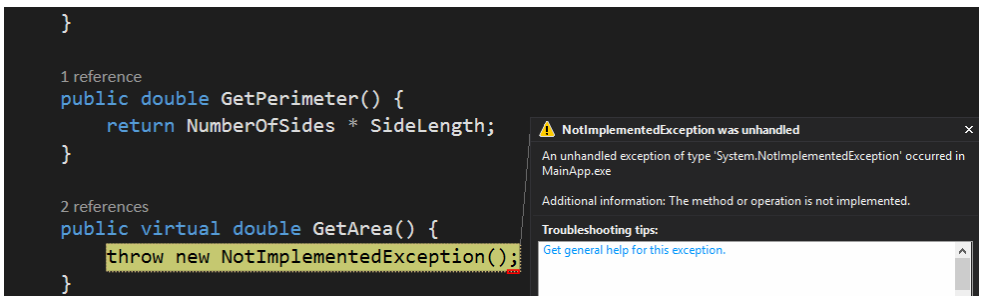
النتيجة سوف تكون صحيحة بعد تشغيل الكود، سوف تجد انه قام بحساب المساحة والمحيط وتم طباعتهم على الشاشة.

الذي يهم في هذه الطريقة هو أن الدالة GetArea في الكلاس (الأب) ConcreteRegularPolygon يقوم بعمل throw exception في حال تم استدعائها، وبالتالي إذا قمت بعمل الكلاس المثلث Triangle ونسيت اعادة تعريف هذه الدالة فسوف تحصل على ال Exception.

للننظر إلى ال class التالي وهو تعريف آخر للمثلث ونسى المبرمج تعريف الدالة GetArea، وتم استخدامه:

```
1.  class Triangle : ConcreteRegularPolygon
2.  {
3.      public Triangle(int length)
4.          : base(3, length)
5.      {
6.      }
7.  }
8.
9.  static void Main(string[] args)
10. {
11.     Triangle sequare = new Triangle(5);
12.     Console.WriteLine("Area: " + sequare.GetArea());
13.     Console.WriteLine("Permieter: " + sequare.GetPerimeter());
14.
15.     Console.ReadKey();
16. }
```

في حال قمت بتشغيل الكود أعلاه سوف تحصل على ال Exception



بالطبع هذا الException نحن الذي كتبناه في كلاس الأب ولكن بسبب أن الابن لم يقوم بتغييره أي عمل Override (**والمترجم لم يجبره على ذلك**) لذلك حدث هذا الخطأ.

فيما يلي سوف نستعرض الحل الثاني في ال Abstraction وهو يحل المشكلة السابقة (المترجم لم يجبر المبرمج على اعادة التعريف) ويجبرك على أن تعيد التعريف override وإلا فلن تتم ترجمة الكود.

تطبيق ال Abstraction باستخدام ال Abstract Class

للتذكير فإن ال Abstract Class هو الكلاس الذي يحتوي على دالة أو أكثر abstract ولا يمكن عمل كائن منه.
الكود التالي يعرض تعريف الشكل بطريقة ال Abstract.

```
1.  public abstract class AbstractRegularPolygon
2.  {
3.      public int NumberOfSides { get; set; }
4.      public int SideLength { get; set; }
5.
6.      public AbstractRegularPolygon(int sides, int length)
7.      {
8.          NumberOfSides = sides;
9.          SideLength = length;
10.     }
11.
12.     public double GetPerimeter()
13.     {
14.         return NumberOfSides * SideLength;
15.     }
16.
17.     public abstract double GetArea();
18. }
```

هذا مشابه للConcrete Class لكن فقط الاختلاف في ال GetArea حيث يوجد تعريف فقط Declaration ولا يوجد body في الكود، أي هي abstract ويجب عمل الكلاس abstract (بالطبع هذا يعني أنه لا يمكن عمل كائن من هذا class).

كلاس المربع والمثلث سوف يكونا متشابهين **وبالطبع** سوف يلزمك المترجم أن تقوم بعمل تعريف للدالة GetArea وإلا فسوف تحصل على خطأ في الترجمة

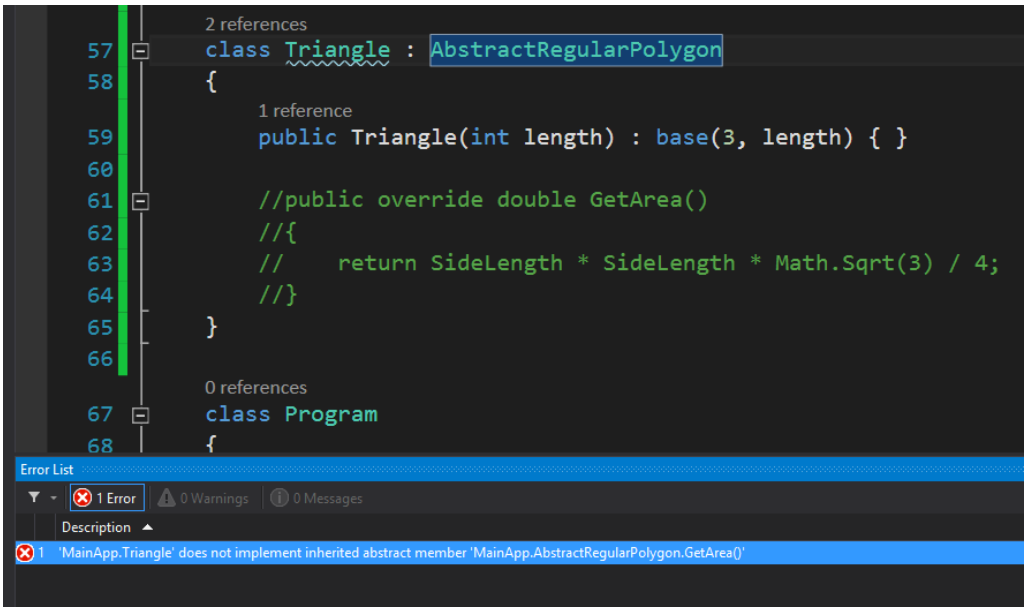
```
1.  public class Square : AbstractRegularPolygon
2.  {
3.      public Square(int length)
4.          : base(4, length)
5.      {
6.      }
7.
8.      public override double GetArea()
9.      {
10.         return SideLength * SideLength;
11.     }
12. }
13. public class Triangle : AbstractRegularPolygon
14. {
```

```
15.     public Triangle(int length)
16.         : base(3, length)
17.     {
18.     }
19.
20.     public override double GetArea()
21.     {
22.         return SideLength * SideLength * Math.Sqrt(3) / 4;
23.     }
24. }
```

الكود في ال main سوف يكون مشابه ولا يوجد عليه تغيير:

```
1.  static void Main(string[] args)
2.  {
3.      Triangle triangle = new Triangle(5);
4.      Console.WriteLine("Area: " + triangle.GetArea());
5.      Console.WriteLine("Permieter: " + triangle.GetPerimeter());
6.
7.      Console.ReadKey();
8.  }
```

جرب أن تقوم بوضع تعليق على أي من دوال ال GetArea في كلاس المثلث أو المربع (على اعتبار أنك نسيت كتابتها) وستجد رسالة الخطأ واضحة، كما يلي:



هكذا بطريقة ال Abstract class فإن المترجم يجبرك على عمل override لكل الAbstract methods وإلا فسوف تحصل على خطأ اثناء عملية الترجمة.Compile Time Error

تطبيق ال Abstraction باستخدام ال Interface

هذه المرة سوف نستخدم ال interface في نفس المثال، وهو مشابه لل abstract class ولكن بدون Implementation (فقط يوضح العقد Contract)، والكود التالي يعرض ال Interface:

```
1.  public interface IRegularPolygon
2.  {
3.      int NumberOfSides { get; set; }
4.      int SideLength { get; set; }
5.      double GetPerimeter();
6.      double GetArea();
7.  }
```

من الConvention في سي# أن تبدأ ال Interface بالحرف ا، ومن العادات الجيدة أن تتبعه في تسميتك، لذلك قمنا بتسميته IRegularPolygon

لاحظ أن:

1. الInterface فقط يحتوي على declaration بدون Implementation، وحتى المتغيرين هم فقط declaration وليس تعريف لمتغير property ويجب أن تكون في الكلاس الذي يطبق ذلك ال Interface
2. عدم وجود أي Access Modifier حيث كل الMembers هم public تلقائياً، وإذا حاولت اضافه أي Modifier سوف تحصل على خطأ في الترجمة

مثال على كلاس المربع الذي يستخدم ذلك الInterface

```
1.  public class Square : IRegularPolygon
2.  {
3.      public int NumberOfSides { get; set; }
4.      public int SideLength { get; set; }
5.
6.      public Square(int length)
7.      {
8.          NumberOfSides = 4;
9.          SideLength = length;
10.     }
11.
12.     public double GetPerimeter()
13.     {
14.         return NumberOfSides * SideLength;
15.     }
16.
17.     public double GetArea()
18.     {
19.         return SideLength * SideLength;
20.     }
21. }
```

لاحظ الآن وجود المتغيرات Automatic Property وهي الImplementation لما هو موجود، بالإضافة إلى الدوال ايضاً.

مثال الmain أيضاً هو نفسه (كالمثال السابق في ال Abstract class أو الاول) ولا يحتاج لأي تغيير مع الأمثلة السابقة.

ملاحظة: إذا حاولت أن لا تطبق كل الدوال الموجودة في ال Interface سوف تحصل على خطأ وقت الترجمة ايضاً (مثل طريقة ال Abstract class).

لنعد لتعديل رقم 2 وهو دالة جلب جميع الأشكال، فسواء قمت بأي طريقة من ال Abstraction السابقة فيمكنك أن تقوم بتطبيق تلك الدالة بسهولة كما يلي:

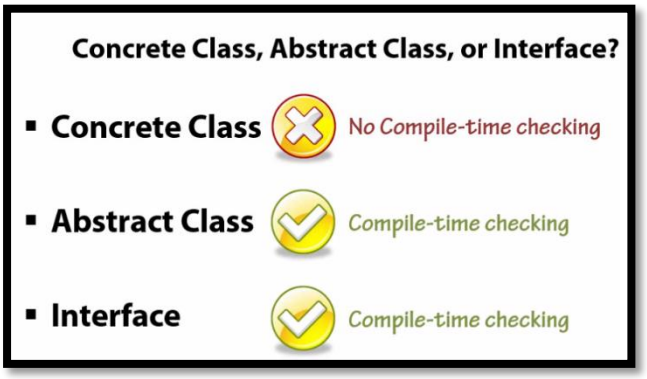
```
1.  private static IRegularPolygon[] ExtractShapes()
2.  {
3.      // Read from external sources and return the polygons
4.      // parsing the file and extract the shapes
5.
6.      return new IRegularPolygon[] {
7.          new Sequare(3),
8.          new Triangle(4),
9.          new Sequare(5)
10.     };
11. }
```

طبعاً بغض النظر عن طريقة ال Parsing والتعامل مع الملف، المهم هو انه استطعنا جلب جميع الاشكال من تلك الدالة، وفي المثال اعلى تم استخدام Interface ك Abstraction. لكن مرة أخرى أي طريقة منهم سوف تفي بالغرض مكانها.

الاختيار بين Concrete Class و Abstract Class و Interface

حتى هذه اللحظة نكون قد عرضنا الحلول الثلاثة لنفس المشكلة والسؤال الآن من تختار فيما بينهم؟

- ال concrete class لا تلزمك بعمل Implementation (وقد تحصل على خطأ على حسب الكود الموجود)
- ال abstract class وال Interface تلزمك بكتابة التعريف Implementation والا سوف تحصل على خطأ وقت الترجمة



طبعاً أن تجد الأخطاء وقت الترجمة أفضل وأسهل لك بكثير من أن تجدها وقت التشغيل، ولذلك فال Concrete Classes ك Abstraction **لا يفضل أن يستخدم**، ويبقى الفرق بين ال Abstract class وال Interface

الفرق بين ال Abstract class وال Interface

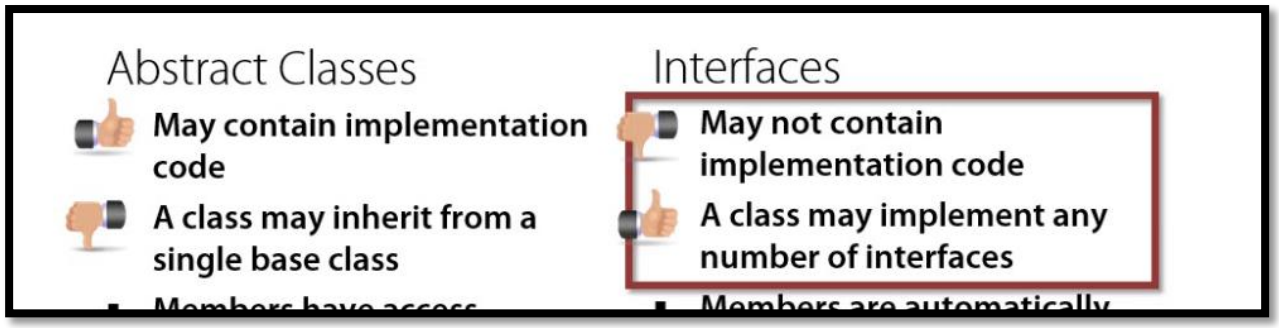
والفرق يكمن في هذه النقاط:

1. ال abstract قد تحتوي على كود Implementation، بينما ال Interface لا تحتوي على كود وانما تحتوي على تصاريح declaration

2. عندما تستخدم ال abstract فإن كل الكلاسات تستطيع الوراثة منه ولغة البرمجة سي# وجافا تسمح بالوراثة فقط من كلاس واحد، لكن ال interface تستطيع عمل Implements لأكثر من interface
3. الأعضاء members في ال abstract class قد تحتوي على Access Modifier، بينما في ال Interface كلها تعتبر Public
4. ال abstract class تحتوي على دوال بناء وهدم ومتغيرات، بينما ال interface لا تحتوي على دالة بناء ولا هدم ولا متغيرات

أول فرقين سوف يعطيك سهولة القرار فيمن سوف تستخدم في تصميمك، فميزة ال Abstract Class سوف يحتوي على الكود الذي تتشارك فيه جميع الأبناء، وهذا مفيد في حالة كان جميع الأبناء سوف يأخذوا نفس ال Implementation لكن مشكلته أنك لا تستطيع أن ترث من أكثر من كلاس واحد لذلك قد لا يجدي معك إذا كان كذلك قد يرث من شيء آخر.

بعكس ال Interface التي يمكن أن تطبق على أكثر من Class، ولكنها بدون Implementation.



نقطة أخرى مهمة في اختيار ال Interface ك Abstraction هي أن الوراثة لا تجدي في كثير من الحالات ويفضل ال Composition عليها، واحياناً تكون خيار خاطئ في حال لم تحقق الشرط IS-A.

في مثالنا الاول (مثال الأشكال) وجدنا أن استخدام ال interface أو ال abstract class يعطي نفس النتيجة، وقد تجد أن ال Abstract مناسب هنا لأن هناك تشارك في الكود (حساب المساحة) بين المثلث والمربع.

في الفصول القادمة سوف تعرف استخدامات انسب لل interface وهكذا سوف يكون لديك كل المعرفة لكي تتخذ القرار المناسب باختيار الinterface والabstract class .

خلاصة الفصل الأول

- ال Interface هي Contract بدون Implementation وتحتوي على Public Members
- المترجم يلزم جميع مستخدمي هذا الInterface بتطبيق كل ال Declaration
- تمت المقارنة بين كل من ال Interface وال Abstract Class ومعرفة ميزة كل منهم.

الفصل الثاني: نظرة حول ال Interface

دور ال Interface في قابلية الصيانة Maintainability

واحدة من أهم الأسباب في استخدام ال Interface هو أنه يساعد في جعل الكود أكثر قابلية للصيانة Maintainable، أي الكود المرن في التغيير وليس الذي يحتاج تغييره كاملاً بسبب تغيير بسيط، وبالتالي يضمن هذا الأمر أن الكود يعمل للمستقبل Future Proof Code.

فالتطبيقات دائماً تتغير إما بتغيير المتطلبات أو بالإضافة المزيد من الخصائص الجديدة، والفكرة هنا هي أنه عندما نريد العمل على جزء واحد من التطبيق فنحن نريد تقليل التغييرات التي سوف تطرأ على بقية الأجزاء في التطبيق. وسوف نرى كيف يمكن أن تفيدنا ال Interface في ذلك.

قبل أن نبدأ، لنرى أحد العادات الجيدة في البرمجة، وال Interface لها نصيب من ذلك، وهي:

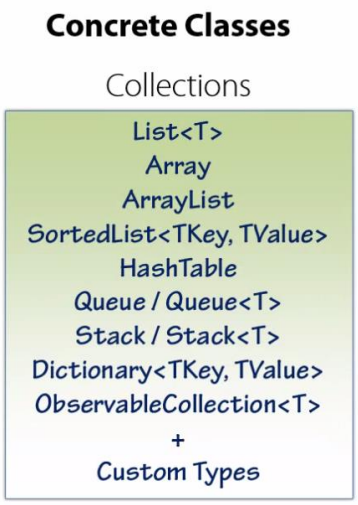
Program to an abstraction rather than a concrete type

كما ذكرنا في الفصل السابق أنه يمكن أن تستخدم ال Interface ك Abstraction وبالتالي تستطيع أن تفهم العادة على أنها:

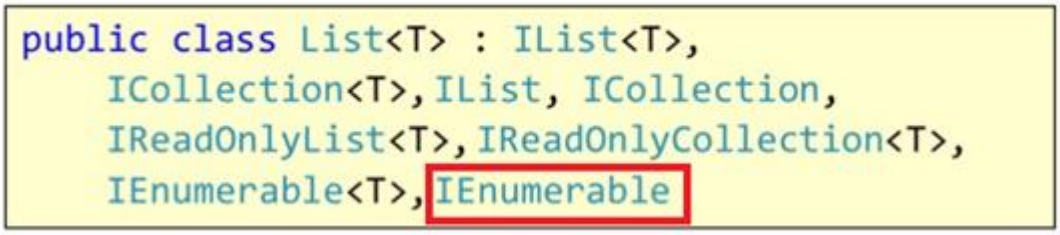
Program to an Interface rather than a concrete type

لذلك عليك البرمجة بال Contract بغض النظر من سوف يطبقه كأن هناك شخص آخر سوف يتولى هذه المهمة.

وتذكر أن ال Concrete Class هي الكلاسات العادية التي تقوم بعملها أو موجودة ضمن مكتبة اللغة أو في مكتبة خارجية أخرى، مثلاً في ال Collections API كل الClasses التالية هي Concrete:



وكل هذه الكلاسات أعلاه تقوم بعمل Implements للكثير من ال Interfaces حتى تقوم بعملها بطريقة جيدة وتقدم Abstraction لك، مثلاً الكلاس List يقوم بعمل Implements لكل ال Interfaces التالية:



لاحظ أن List Class يطبق كل هذه ال Interfaces (بدلاً من وضعها في واحدة كبيرة، وهذا مثال ل Interface Segaration Principle والفكرة أن الInterface يستطيع أن يرث من أي Interface أخرى وبالتالي لا تحتاج لتعديل أي Interface بعد أن يخرج للProduction وصار الكثير من الclients يعتمدوا عليها، راجع الملحق 2 للتعرف على فكرة الوراثة من الInterface).

ال IEnumerable أحد ال Interface التي تطبقها ال List والكثير من ال Concrete Collection وهي تستخدم عندما تريد الloop على الcontainer ، ولذلك لتطبيق القاعدة Program to abstraction فيمكنك استخدام ال interface بدلاً من اسم الكلاس عندما تستخدم أي من ال Containers.

الفرق بين Programming to Concrete و Programming to Abstraction

لو لاحظت لدالة ارجاع الاشكال ExtractShapes في الفصل السابق، كانت بهذا الشكل:

```
1.  static void Main(string[] args)
2.  {
3.      IRegularPolygon[] shapes = ExtractShapes();
4.      foreach (IRegularPolygon shape in shapes)
5.      {
6.          Console.WriteLine("Area: " + shape.GetArea());
7.      }
8.
9.      Console.ReadKey();
10. }
11.
12. private static IRegularPolygon[] ExtractShapes()
13. {
14.     // Read from external sources and return the polygons
15.     // parsing the file and extract the shapes
16.
17.     return new IRegularPolygon[] {
18.         new Sequare(3),
19.         new Triangle(4),
20.         new Sequare(5)
21.     };
22. }
```

طريقة استخراج الأشكال خارج نطاق الموضوع، لذلك افترض أنك حصلت عليهم سواء من قراءتك لملف أو من قاعدة بيانات أو أي مصدر يكن، ومن ثم ترجع تلك الأشكال.

الذي يهم هنا طريقة ارجاع المجموعة ولاحظ أنه تم ارجاعها على شكل مصفوفة (وهي تعتبر Concrete Type)، فماذا يحدث لو قمت بتغيير الكود ورجعت Data Structure أفضل منها، مثلاً List أو Set لأنك لا تريد تكرار تلك الأشكال التي لها نفس الطول مثلاً؟

لذلك حتى تطبق تلك العادة عليك بإرجاع ال Abstraction (باستخدام ال IEnumerable) وبالتالي تضمن أن أي Implementation يطبق ال IEnumerable سوف يعمل بغض النظر عن ماهيته والسبب كونك تعتمد على ذلك ال Abstraction. نلاحظ الكود الآن بعد التغيير إلى IEnumerable<IRegularPolygon>

```
static void Main(string[] args)
{
    IEnumerable<IRegularPolygon> shapes = ExtractShapes();
    foreach (IRegularPolygon shape in shapes)
    {
        Console.WriteLine("Area: " + shape.GetArea());
    }

    Console.ReadKey();
}

private static IEnumerable<IRegularPolygon> ExtractShapes()
{
    // Read from external sources and return the polygons
    // parsing the file and extract the shapes

    return new IRegularPolygon[] {
        new Sequare(3),
        new Triangle(4),
        new Sequare(5)
    };
}
```

الكود سوف يعمل كمخرج بدون أي تغيير، ولكن الآن جرب غير الكود داخل الدالة وبدل أن ترجع Array جرب أن ترجع List أو Queue، وستجد أن ال Main (ال Client Code) لم تتغير ابداً، وهكذا تستطيع تغيير ال Implementation طالما يطبق ال Abstraction بدون تغيير ال client code، وهذا ما أطلقنا عليه اول الفصل بالكود المستقبلي Future Proofing code.

طرق أخرى في ال Maintainability

استخدام ال Interface ليس الطريق الوحيدة لجعل الكود أكثر قابلية للصيانة، فهذه الخاصية لها أكثر من مسار تستطيع تطبيقه حتى يكون كودك أفضل، مثلاً:

- جعلته أكثر مقروئية Readability من ناحية الأسماء والتنظيم وعدم استخدام الدوال والكلاسات الطويلة.
- جعل الدوال والكلاسات لديك مسؤولة عن شيء واحد فقط Single Responsibility وهكذا لن تكون مضطر لتغيير تلك الأمور إلا لسبب واحد فقط
- كلما حولت مشروعك لفكرة ال Layers (Modules أو Packages أو Namespaces) وقللت التداخل Coupling بين تلك الوحدات وأخفيت كل شيء بقدر الامكان Hide Implementation أي جعلته private، لذلك النصيحة العامة لا تستخدم public إلا عندما تحتاجها فقط. سوف يتم تناول موضوع ال Layering في أحد الفصول القادمة.

المقروئية ليست في التعليقات!

المقروئية Readability لا تكون من خلال التعليقات فقط والتي يفضل أن تعتبرها أخر ورقة لديك تقوم بها عندما لا تستطيع كتابه كود أفضل من الذي قمت بكتابته. خصوصاً لو كان مجرد Business Logic وليس عباره عن خوارزمية ما (مثلاً حساب تقويم أو معالجة لصورة بطريقة ما).

وقد تصل للمقروئية الجيدة في حال اتبعت التسميات الجيدة والتي فيها الغالب لكل لغة Convention خاص بها، وأيضاً إذا استخدمت بعض ال Patterns لكي تحسن الAPI لديك، على سبيل المثال لماذا تقوم بعمل دالة بناء Constructor تستقبل 20 متغير؟ فيمكنك بدلاً من ذلك أن تستخدم Builder Pattern أو Static Factory وذلك لتحسين الAPI .

الفصل القادم سوف نرى أحد اهم الأسباب لاستخدام ال Abstraction وبالأخص ال Interface وهي في جعل الكود قابل للتطوير Extendibility.

خلاصة

- تحدثنا عن مفهوم Program to abstraction rather than a concrete type والذي يجعل الكود أكثر صموداً مع التغيير، وليس كما هو الحال مع ال concrete type والذي يحدث فيه Broken بسهولة وقد تحتاج لعمل التغييرات.
- ال Interface يبعدك عن تفاصيل ال Implementation وتجعلك تركز على ال Contract الموجود.
- يفضل بدل ارجاع list أو Array أن تقوم بإرجاع IEnumerable، لأن الراجع قد يتغير من مصفوفه إلى list وسوف تحتاج لتعديل ال client ولكن عندما تستخدم ال interface فلن يحتاج ال client للتغير لأن الاثنين يطبقوا ال IEnumerable وهكذا لكل API التي تقوم بعملها.

الفصل الثالث: استخدام الInterface بكفاءة

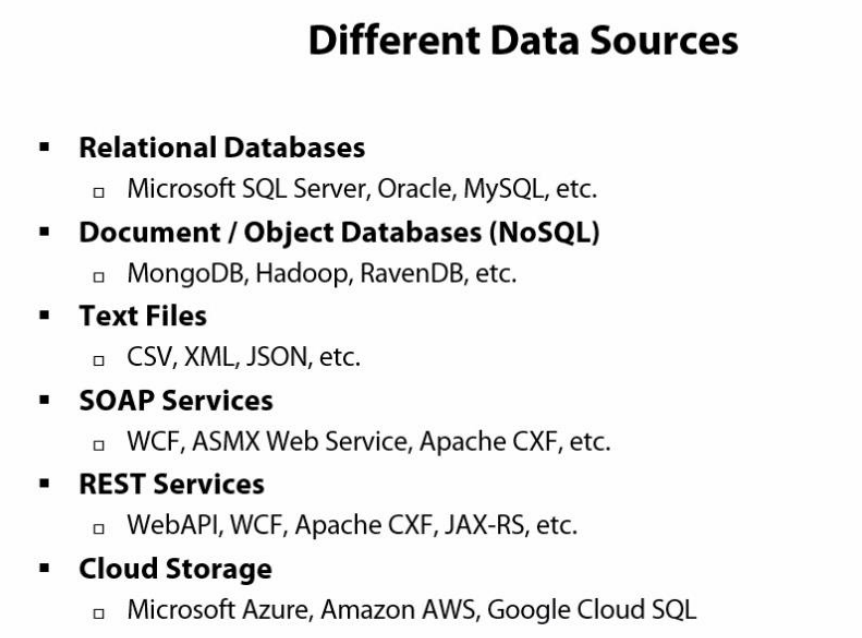
دور ال Interface في ال Extensibility

سوف نرى في هذا الفصل كيف يمكن أن تستخدم ال Interface في جعل الكود قابل للتطوير بسهولة، حيث نريد انشاء الكود ال Extensible أي القابل للتعديل بسرعه بعد تغيير المتطلبات، فاذا تم عمل ال Contract وجعل الApplication يستخدمه فيمكن استخدام أي Implementation لهذا العقد.

مثال عملي يوضح فائدة ال Interface

مثلاً لدينا تطبيق نريد تشغيله لدى أكثر من جهاز، وكل جهاز سوف يتم تخزين/قراءة البيانات بطريقة مختلفة (سواء في قاعدة بيانات مختلفة أو في مكان مختلف) أي different data store، وبالتالي نريد بناء التطبيق بحيث يعمل مع هذه ال data store / storages المختلفة

الصورة التالية تبين انواع ال data stores التي يمكن أن تستخدم، بدئاً من قواعد البيانات العلائقية وهناك عدة انظمة فيها، ومروراً بقاعدة بيانات ال NO SQL والملفات النصية وبخدمات الويب سواء SOAP أو RESTful وانتهاءً بتخزين البيانات على ال Cloud، وكل هذه انواع من طرق التخزين التي يمكن تستخدم.



البرنامج سوف يقرأ معلومات المستخدمين Persons وهي الاسم الأول والثاني وبضعه معلومات لكل مستخدم، بالإضافة إلى امكانية اضافة مستخدم جديد، حذف مستخدم، تحديث بيانات مستخدم. هذا العمليات تسمى CRUD وسوف نبين هذا الاختصار بعد قليل.

حالياً للتسهيل سوف يكون المطلوب أن يعمل البرنامج على ثلاثة أنواع من ال data sources:

- قاعدة بيانات SQL Server
- ملفات نصية من نوع CSV
- خدمات الويب من نوع SOAP

بدون استخدام أي تصميم للكود، فيمكن كتابته مباشرة عن طريق جمل ال If وفحص الطريقة المطلوبة بطريقة تشابه ما يلي:

```
1.  public IEnumerable<Person> ReadData(int source) {
2.      List<Person> persons = new List<Person>();
3.
4.      if (source == 1) {
5.          // Read from SQL Server Database
6.      }
7.      else if (source == 2) {
8.          // Read from CSV File
9.      }
10.     else if (source == 3) {
11.         // Read from SOAP Web Service
12.     }
13.
14.     return persons;
15. }
```

بعد أن قمت بعمل كلاس يمثل ال Person واتبعت نصيحة Programming to Abstraction قمت بعمل ارجاع للIEnumerable ، ولكن داخل الدالة قمت بعمل فحص للقيمة التي تقرأها من المستخدم أو من ملف خارجي وتقوم على اساسها بقراءة البيانات من المكان المطلوبة.

الكود السابق به العديد من المشاكل:

- في حال اضفت أي source جديد سوف تحتاج تضيفها في هذه الدالة، وتضيفها ايضاً في بقية الدوال التي تخزن وتحذف وتحدث لأنها سوف يكون بها نفس الشرط وسوف تكون مكررة في مواضع أخرى.
- الدالة سوف تصبح كبيرة وبالتالي لها مسؤوليات كثيرة

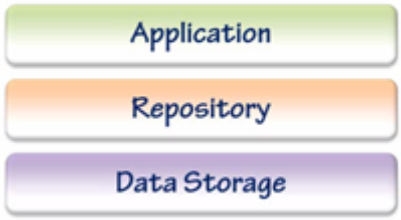
الحل المناسب والذي يعد من أسهل طريقة للعمل مع عدة data sources مختلفة هو باستخدام الpattern المعروف بالاسم Repository Pattern والذي يستخدم ال interface بشكل اساسي.

ال Repository Pattern

وهو من طرق التصميم المعروفة Design Pattern ويستخدم لأضافه طبقة من ال Layer of Abstraction، وهذا تعريفه من الكتاب المعروف Patterns of Enterprise Application Architecture للمؤلف Martin Fowler:

Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects.

لتبسيط التعريف: وهو طبقة وسيطة بين البرنامج وآلية التخزين، وبالتالي كود التطبيق يتعامل مع هذه الطبقة بدلاً من أن يعتمد على الية التخزين مباشرة والتي يمكن أن تتغير، كما في الصورة التالية سوف تجد أن طبقة التخزين Repository الآن وسيطة بين الكود وبين قاعدة البيانات.



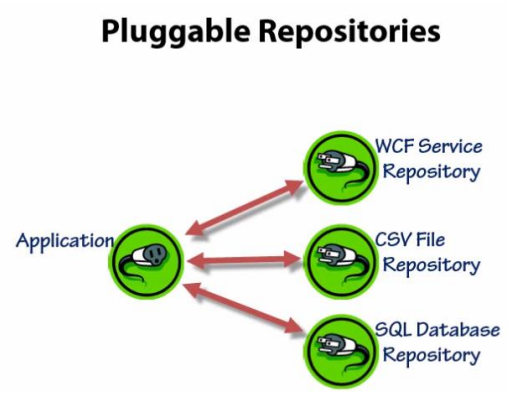
فنحن لا نريد التطبيق أن يتصل مباشرة بالقاعدة أوData source (لأن في تلك الحالة التطبيق يجب أن يعرف كيف سيتعامل مع نوع ال data source المعين مثلاً عمل SQL Query أو القيام بال Web service Call)، وبدلاً من ذلك سوف نضيف طبقة Layer بين التطبيق وال data source وهذه هي ال Repository، وسوف يتصل التطبيق بها بدون أن يعرف كيف ستتعامل ال repository مع ال data source، بعباره أخرى التطبيق فقط سوف ينادي ال contract المطبق في ال repository وهو لا يهتم بأي Implementation.

عملياً قد تكون هناك layers بين ال Application (وهي هنا UI Layer) وال Repository (مثلاً domain layer أو BLL) وهي تتعامل مع ال Repository ولكن للتبسيط ولتوضيح فائدة الinterface في ال Extensibility سوف تعمل ال Application مباشرة مع ال Repository ونتجاهل بعض ال Layers، والفصل الخامس سوف نأخذ مثالاً على ال Layers.

اذاً: التطبيق سوف يعتمد على ال interface وهو يتوقع أي Implementation موجود أن يطبقها قبل أن يتعامل معه، بغض النظر كونه:

- WCF Service Repository للتعامل مع SOAP service
- أو CSV Service Repository للتعامل مع الTEXT FILE
- أو SQL Service Repository للتعامل مع RDBMS

أو أي نوع آخر (مثلاً يتعامل مع ال Azure SQL) طالما يطبق نفس ال Interface، فكل شيء سيعمل مباشرة بدون تغيير أي كود



إذا لم تضح لك الصورة بعد، فلا تقلق وسوف تتضح لك كل الأمور مع المثال، لنوضح الآن ماذا نقصد بال CRUD

ماذا نعني بال CRUD

هي مجموعه من الدوال طالما تكون موجودة في أي تطبيق يريد التعامل مع أي جدول في القاعدة أو عموماً مع أي Data Storage، وال CRUD هي اختصار للعمليات الإضافة Create، القراءة Read، التحديث Update، الحذف Delete.

مثال ال Repository باستخدام ال Interface

لنبدأ الآن بالبرمجة، وسوف نقوم بتعريف المستخدم Person

```
1.  public class Person
2.  {
3.      public int Id { get; set; }
4.      public string FirstName { get; set; }
5.      public string LastName { get; set; }
6.      public DateTime BirthDate { get; set; }
7.  }
```

بعد ذلك سوف نقوم بعمل Repository لكي يتعامل مع ال CRUD الخاص بال Person وسوف نعرف ال Contract كما يلي:

```
1.  public interface IPersonRepository
2.  {
3.      void AddPerson(Person person);
4.      IEnumerable<Person> GetPeople();
5.      Person GetPerson(string firstName);
6.      void UpdatePerson(int personId, Person updatedPerson);
7.      void DeletePerson(string firstName);
8.      void UpdatePeople(IEnumerable<Person> updatedPeopele);
9.  }
```

هنا أنشئنا ال IPersonRepository وفيها دالة للإضافة، ودالة للقراءة سواء لشخص أو مجموعه، وأيضاً دالة للتحديث، وأخيراً دالة لحذف الشخص، وهكذا نقوم بعمل كل ال operations التي نريدها على البيانات.

لاحظ العائد من دالة ارجاع الأشخاص هو IEnumerable وبالتالي كل الكلاسات التي تطبق هذه الInterface تستطيع ارجاع Array،List أو أي كائن يطبق ال IEnumerable، ونفس الامر في دالة التحديث والتي يتم تمرير مجموعه من الأشخاص (وهنا نطبق ال Program to abstraction).

الآن نريد أن نتعامل مع أكثر من Data Source والبرنامج سوف يجلب البيانات من:

1- جلب من ال Service Repository
2- جلب من ال CSV Repository
3- جلب من SQL Repository

سوف يتم وضع تعريف الكود الخاص بجلب ال People أي دالة واحدة GetPeople فقط في كل كلاس حتى نركز على الفكرة العامة وليس على حول كيف يمكن ادخال أو حذف البيانات من CSV أو Service.

التعامل مع الملفات CSV Repository

سوف نقوم بعمل كلاس جديد وليكن اسمه CVSRepository لكي يتم جلب البيانات من الملف، بالطبع هذا الكلاس سوف يطبق ال IPersonRepository كما يلي:

```
1.  public class CSVRepository: IPersonRepository
2.  {
3.      public void AddPerson(Person person)
4.      {
5.          throw new NotImplementedException();
6.      }
7.
8.      public IEnumerable<Person> GetPeople()
9.      {
10.         List<Person> persons = new List<Person>();
11.         // fill persons from file
12.         return persons;
13.     }
14.
15.     public Person GetPerson(string firstName)
16.     {
17.         throw new NotImplementedException();
18.     }
19.
20.     public void UpdatePerson(int personId, Person updatedPerson)
21.     {
22.         throw new NotImplementedException();
23.     }
24.
25.     public void DeletePerson(string firstName)
26.     {
27.         throw new NotImplementedException();
28.     }
29.
30.     public void UpdatePeople(IEnumerable<Person> updatedPeopele)
31.     {
32.         throw new NotImplementedException();
33.     }
34. }
```

سوف نقوم بكتابة كود جلب البيانات من الملف، والملف اسمه data.txt وبنيته بهذا الشكل:

```
1, Wajdy, Essam, 24/08/1985
2, Ali, Salem, 01/01/1980
3, Ayman, Ahmed, 05/05/1990
```

سوف يكون كود استخراج البيانات منها كالتالي (لا داعي للتركيز حول الكيفية وليست هي الطريقة الأفضل):

```
1.  public IEnumerable<Person> GetPeople()
2.  {
3.      List<Person> people = new List<Person>();
4.
5.      string[] lines = File.ReadAllLines("data.txt");
6.      foreach (string line in lines)
7.      {
8.          string[] tokens = line.Split(',');
9.
10.         Person person = new Person
11.         {
12.             Id = int.Parse(tokens[0].Trim()),
13.             FirstName = tokens[1].Trim(),
14.             LastName = tokens[2].Trim(),
15.             BirthDate = DateTime.ParseExact(tokens[3].Trim(), "dd/MM/yyyy",
16.                 CultureInfo.InvariantCulture)
17.         };
18.
19.         people.Add(person);
20.     }
21.
22.     return people;
23. }
```

بهذا الشكل سوف يتم تعريف بقية الدوال الموجودة في ال CSVRepository وبالتالي يستطيع هذا الكلاس اضافة، عرض، تحديث، حذف الأشخاص من الملف (يمكنك تطبيقها كتمرين في البرمجة).

التعامل مع قاعدة البيانات SQL Repository

سوف يتم الآن تطبيق الكلاس الذي يتعامل مع قاعدة البيانات ويقوم بنفس المهام ولكن التخزين سوف يكون على القاعدة. وسنقوم بعمل Implement لل Interface والتركيز على دالة GetPeople ويبقى تطبيق البقية اليك كتمرين أيضاً.

ولكتابة الدالة التي تجلب من قاعدة البيانات يمكن أن نستخدم ال ADO.NET ونقوم بفتح الاتصال والاستعلام بجمل SQL وجلب البيانات أو يمكن أن نستخدم أي من ال ORMs الموجودة مثلاً LINQ To SQL أو Entity Framework، وفي هذا المثال سوف نستخدم الطريقة التقليدية.

```
1.  public IEnumerable<Person> GetPeople()
2.  {
3.      List<Person> people = new List<Person>();
4.      using (SqlConnection connection =
5.       new SqlConnection(ConfigurationManager.ConnectionStrings["PeopleDB"].ToString()))
6.      {
7.          connection.Open();
8.          using (SqlCommand cmd = new SqlCommand("SELECT * FROM People", connection))
9.          {
10.             using (SqlDataReader reader = cmd.ExecuteReader())
11.             {
12.                 if (reader != null)
13.                 {
14.                     while (reader.Read())
15.                     {
16.                         Person person = new Person()
17.                         {
```

```
                                Id = int.Parse(reader["Id"].ToString()),
                                FirstName = reader["FirstName"].ToString(),
                                LastName = reader["LastName"].ToString(),
                                BirthDate = DateTime.Parse(reader["BirthDate"].ToString())
                            };

                            people.Add(person);
                        }
                    }
                }
            }
        }

        return people;
    }
```

يتم جلب البيانات من قاعدة البيانات التي تم تحديد ال Connection String لها في ملف ال App.config:

```
<connectionStrings>
  <add name="PeopleDB"
    connectionString="Data Source=.;Initial Catalog=PeopleDB;Integrated Security=True"
    providerName="System.Data.SqlClient"/>
</connectionStrings>
```

المهم حالياً أن القاعدة PeopleDB موجودة وبها جدول اسمه People والبيانات التي توجد بها:

WAJDYESSAM-PC.Pe...eDB - dbo.People

	Id	FirstName	LastName	BirthDate
	2	Abdullah	Mohammed	1987-05-08 00:00:00.000
	3	Osama	Naif	1970-08-07 00:00:00.000
▶*	NULL	NULL	NULL	NULL

بقية الدوال الأخرى يمكنك أن تقوم بتطبيقها ايضاً (للحذف والتعديل والاضافة).

التعامل مع الويب سيرفس Service Repository

هناك أنواع كثيرة لتطبيق الويب سيرفس في ال NET.، فمثلاً يمكن أن تكون WCF أو Web API أو حتى ASMX، ولكن الهدف من هذا الموضوع هو توضيح كيف أن ال Interfaces يجعلك تتعامل مع الجميع بطريقة واحدة وليس شرح كيفية عمل الWeb service ، لذلك سوف يتم جلب البيانات من List في الذاكرة فقط.

للنظر الأن لتعريف ال Service Repository ولاحظ أنها ارجعت بيانات ثابتة على فرض أنها تأتي من خدمة ويب.

```
public IEnumerable<Person> GetPeople()
{
    // simulate call to web service then retreive the result
    return new List<Person> {
        new Person {Id = 1, FirstName = "Bahi",
            LastName="Salem", BirthDate=DateTime.Now},
        new Person {Id = 2, FirstName = "Omar",
            LastName="Omar", BirthDate=DateTime.Now},
```

```
9.          new Person {Id = 3, FirstName = "Khalid",
10.             LastName="Nour", BirthDate=DateTime.Now},
11.         new Person {Id = 4, FirstName = "Ahmed",
12.             LastName="Essa", BirthDate=DateTime.Now}
13.     };
14. }
```

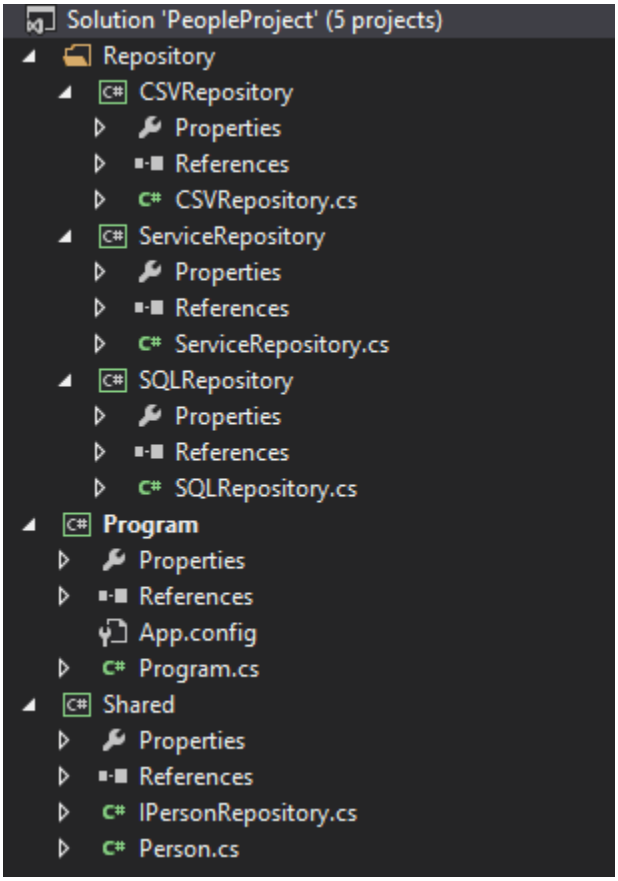
العمل على ال Solution

انتهينا من كتابة ال Repositories الثلاثة، لكن قبل المضي في استخدامها، سوف نقوم بوضع كل منهم على Project منفصل (من نوع Class Library Project) ومن ثم نضيف ال Assembly الخاصة بكل مشروع إلى البرنامج الذي يحتوي على الدالة main.

لمن لم يستخدم ال Class Project من قبل أو يحتاج لعمل مراجعة فننصح بمراجعة الملحق 1 الآن، وبعدها يكمل هنا، حيث به بعض المعلومات المهمة حول المكتبات وكيفية عملها ومتى تحمل للذاكرة.

ال Class Library هو مشروع ولكن ليس له أي مخرج وانما سيخرج ملف DLL تستطيع استخدامه في مشاريع اخرى، ودائماً يفضل أن تقسم المشروع إلى عدة اقسام مثلاً Class Library Project يحتوي على الكود الخاص بالتعامل مع قواعد البيانات، ومشروع Class Library Project آخر يحتوي على الأشياء العامة التي تحتاجها في مواضع أخرى مثلاً ارسال البريد، عمل معالجة للنصوص، وهكذا تستطيع اعادة استخدامها بسهولة من مشروع لأخر بدون الحاجة لإعادة كتابة نفس الكود مرة اخرى أو القيام بعمليات النسخ واللصق. الفصل الخامس سوف نتحدث بمثال عملي عن التقسيم وال Layers.

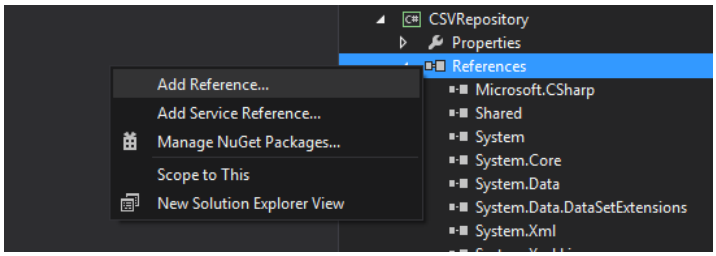
شكل المشروع بعد عمل عدة مشاريع Class Libraries سوف يكون كالتالي:



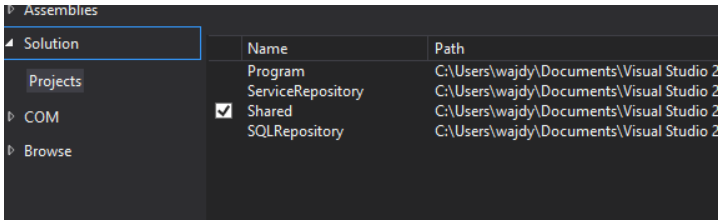
- اولاً المشروع Program وهو Console Application والذي فيه الدالة الرئيسية Main ومنه يعمل البرنامج. ولاحظ أنه Startup Project (تستطيع اختيار أي مشروع بالزر الايمن وتحديد أنه Startup Project، وبالطبع يجب أن يكون Console Project أو Web Project والا فلن يعمل إذا كان Class Library Project)
- المشروع Shared ووضعنا فيه ال Interface بالإضافة إلى الكلاس Person وهي الاشياء المشتركة أو الكلاسات التي تمثل ال Domain في المشروع.
- مجلد Repository ويحتوي على 3 مشاريع وهي التي تعمل على ال Data Sources المختلفة، وتم وضعهم داخل المجلد للتنظيم فقط لا أكثر.

كل من هذه المشاريع الثلاثة سوف تحتاج الوصول إلى ال IPersonRepository وال Person حتى تعمل صحيحاً، لذلك سوف نضيف مشروع ال Shared إلى كل من هذه المشاريع الثلاثة والخطوات كالتالي:

لنبدأ بمشروع CSV والبقية ستكون بنفس الخطوات، سوف تجد قائمة ال References وهي المكتبات التي يتعامل معها هذا المشروع، قم بفتحها واختر Add Reference ثم اذهب ال Solution على اليسار واختر Shared (كما في الصورة التي تليها) وهو المشروع الذي نريد أن يوصل لها ال CSVRepository اليه، وحدده ومن ثم اضغط على OK



هنا لتحديد المشروع:



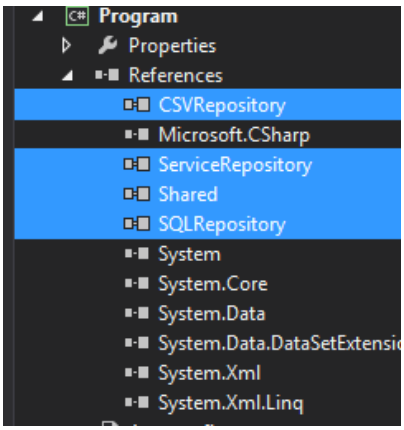
معلومة: تستطيع اضافة المكتبات عموماً (بالأصح ال Assemblies) بعدة طرق منها:

- ال Solution في حال كانت المكتبة في نفس المشروع (كما هو في مثالنا الآن)
- أو من خلال ال Assemblies وهي المتوفرة في .NET. حيث في الوضع الافتراضي لا يتم تحميلها كلها في أي مشروع وانما بضعه Assemblies مهمة
- أو تستطيع تحديد مسارها Browse من جهاز
- أو استخدام ال Nuget Package Manager وهو الأسلوب الأفضل خصوصاً في المكتبات الخارجية Third-Party Libraries.

حالياً سوف نضع ال Shared Assembly في المشروعين الباقين وهم SQLRepo وال ServiceRepo بنفس الطريقة اعلاه.

أخيراً سوف نضيف ال Assembly الخاصة بالمشاريع الثلاثة SQLRepo،ServiceRepo، CSVRepo إلى المشروع Program حتى نستطيع استخدامهم جميعاً عند تشغيل البرنامج.

وكما يتبين الآن أن ال Program يستخدم تلك المكتبات



كود الدالة الرئيسية Main

سوف نسأل المستخدم عما يريده أولاً، ومن ثم بناءً على اختيار المستخدم سوف نقوم بطباعة معلومات الأشخاص ونقوم بإنشاء ال Repository المناسب:

```
1.  class Program
2.  {
3.      static void Main(string[] args)
4.      {
5.          Console.Write("Enter your choice: ");
6.          int choice = Convert.ToInt32(Console.ReadLine());
7.
8.          if (choice == 1) {
9.              IPersonRepository repo  = new CSVRepository();
10.             IEnumerable<Person> people = repo.GetPeople();
11.             displayPeople(people);
12.         }
13.         else if (choice == 2) {
14.             IPersonRepository repo  = new SQLRepository();
15.             IEnumerable<Person> people = repo.GetPeople();
16.             displayPeople(people);
17.         }
18.         else if (choice == 3){
19.             IPersonRepository repo  = new ServiceRepository();
20.             IEnumerable<Person> people = repo.GetPeople();
21.             displayPeople(people);
22.         }
23.
24.         Console.ReadKey();
25.     }
26.
27.     private static void displayPeople(IEnumerable<Person> people)
28.     {
29.         foreach (var person in people)
30.         {
31.             Console.WriteLine(person.Id + " " + person.FirstName +
32.                 " " + person.LastName);
33.         }
```

```
    }
}
```

الآن لو جربت البرنامج سوف تجد أن الخيار يطبع البيانات التي توجد على ال CSV بينما 2 يطبع على ال SQL و3 يطبع التي توجد على ال Service. وهكذا توحدت ال API ولكن ال Implementation يختلف على حسب نوع ال Repository. طبعاً سوف يكون هذا الأمر مطبق على جميع الدوال وليس فقط GetPeople.

وبهذا الشكل فإن المشروع يتعامل مع هذه ال Repositories الثلاثة، وكل ذلك من خلال ال Interface، **رائع**!

حذف الكود المكرر باستخدام ال Factory Method

لاحظ وجود الكود المكرر داخل جمل ال IF في الكود السابق، والكود متشابه فقط الإختلاف في سطر عمل ال Implementation وهو شيء واحد فقط يتغير في الثلاثة دوال، لذلك كما قلنا سابقاً هناك تصميم غير جيد، ويجب التحسين Refactoring، وسوف نقوم بعمل Refactor لهذه الأكواد وبالتالي يتم مسح التكرار، وسوف نستخدم Factory Method للقيام بذلك.

سوف نضيف كلاس Repository Factory وفيه دالة ترجع نوع ال IPersonRepsotiry وتأخذ قيمة نوع ال repo المطلوب وباستخدام جملة if أو switch يتم ارجاع الImplementation المطلوب، وفي حال أرسلت نوع خاطئ سوف يتم عملthrow لل exception كما في الشكل التالي:

للاستزادة عن ال Static Factory Method يمكن قراءة هذه المقالة وهي بالجافا ولكن المبدأ نفسه هل مللت من دالة البناء Constructor؟

```
public class RepositoryFactory
{
    public static IPersonRepository GetRepository(string type)
    {
        IPersonRepository repo = null;

        switch (type)
        {
            case "Service":
                repo = new ServiceRepository();
                break;

            case "SQL":
                repo = new SQLRepository();
                break;

            case "CSV":
                repo = new CSVRepository();
                break;

            default:
                throw new ArgumentException("Invalid Repository Type");
        }

        return repo;
    }
}
```

الآن الدالة الرئيسية Main سوف تكون بهذا الشكل:

```
1.  class Program
2.  {
3.      static void Main(string[] args)
4.      {
5.          Console.Write("Enter your choice: ");
6.          int choice = Convert.ToInt32(Console.ReadLine());
7.
8.          if (choice== 1)
9.              FetchData("CSV");
10.         else if (choice == 2)
11.             FetchData("SQL");
12.         else if (choice == 3)
13.             FetchData("Service");
14.
15.         Console.ReadKey();
16.     }
17.
18.     private static void FetchData(string type)
19.     {
20.         IPersonRepository repo = RepositoryFactory.GetRepository(type);
21.         IEnumerable<Person> people = repo.GetPeople();
22.         displayPeople(people);
23.     }
24.
25.     private static void displayPeople(IEnumerable<Person> people)
26.     {
27.         foreach (var person in people)
28.         {
29.             Console.WriteLine(person.Id + " " +
                        person.FirstName + " " + person.LastName);
30.         }
31.     }
32. }
```

كل شيء يعمل كما كان بالضبط، ولكن بكود أفضل وبدون تكرار. رائع**!**

الدالة FetchData لا تهتم بنوع ال type الذي يرجعه ال Factory سواء كان csv أو sql فهذا الكود يعمل بنفس الطريقة (لأنه يعمل باستخدام ال contract).

أيضاً تستطيع إضافة أي repo جديد بسهولة، فقط سوف تغير في ال factory وتضيف النوع الجديد الذي ترغب فيه واعادة ترجمة المشروع. وهكذا بال interface تحقق فكرة سهولة ال Extensibility.

فقط هناك عيب بسيط، ماذا لو كنا نريد أن يعمل البرنامج على Repository واحد فقط، ومن خلال تغيير في ملف خارجي Configuration يتم تحديد نوع ال Repository، مثلاً في جهاز العميل الأول سوف يعمل البرنامج باستخدام ال CSV File ويخزن ويستعرض البيانات. بينما نفس البرنامج يعمل باستخدام ال SQL Database فقط بتغيير اسم ال Repository في ملف ال Configuration بدون تغيير أي جزئية في الكود وإعادة ترجمة المشروع؟

بمعنى أن تكون الدالة الرئيسية بهذا الشكل

```
1.  class Program
2.  {
3.      static void Main(string[] args)
```

```
    {
        FetchData();
        Console.ReadKey();
    }

    private static void FetchData()
    {
        IPersonRepository repo = RepositoryFactory.GetRepository();
        IEnumerable<Person> people = repo.GetPeople();
        displayPeople(people);
    }

    private static void displayPeople(IEnumerable<Person> people)
    {
        foreach (var person in people)
        {
            Console.WriteLine(person.Id + " " +
                   person.FirstName + " " + person.LastName);
        }
    }
}
```

وعلى حسب القيمة التي توجد في ال Configuration File يتم استخدام ال Repository المناسب.

بالإمكان عمل ذلك عن طريق اضافة فكرة ال Dynamic Loading وبالتالي لا تحتاج لعمل hard coded لل Implementations في ال Factory Method، وسيتم تحميل ال Implementation تلقائياً وقت التشغيل إذا أردت على حسب ما يتم تحديده في الملف.

خلاصة

- تعلمنا مفهوم ال Repository pattern وما هي ال CRUD
- تعلمنا كيف تقوم بعمل custom interface وتقوم بتطبيقه في عدة أماكن
- لإنشاء ال Repository
  - يمكن استخدام ال Factory لأنشاء ال Repository بطريقة ال hard coded أي Compile time
  - أو من خلال ال Dynamic loading لل Repository في وقت التشغيل كما سيأتي الفصل المقبل.

الفصل الرابع: ال Dynamic Factory

قمنا سابقاً باستخدام عدة Implementations لل Interface (سواء في ال Factory class أو حتى عندما كانت جميعها داخل ال main program) وعندما نستخدمها بهذا الشكل فهي تعتبر Compile Time Reference لل Concrete Classes (بمعنى أي اضافة او حذف ل Implementation يحتاج اعادة الترجمة).

لكن يمكن تحسين الأمر، حيث يمكن تغيير البرنامج وجعله لا يعلم أي شيء عن ال Concrete classes في وقت الترجمة Compile Time ولا تحتاج لإضافة ال References لل Implementation في المشروع، وبدلاً عن ذلك نقوم بتحميل النوع الذي نريده عندما يعمل التطبيق dynamic loading concrete type، وهكذا نقوم بحذف أي اتصال Connection بين البرنامج مع أي Implementation (يمكن أن نسميه ال Ultimate Decupling) وهذه تكملة لفوائد ال Interface في ال Extensibility

أخر مرة قمنا فيها في كتابة الكود كنا نطبق مفهوم Programming to Abstraction rather than implementation، ولننظر للدالة FetchData

```
1.  private static void FetchData(string type)
2.  {
3.      IPersonRepository repo = RepositoryFactory.GetRepository(type);
4.      IEnumerable<Person> people = repo.GetPeople();
5.      displayPeople(people);
6.  }
```

لاحظ أننا نتعامل مع ال Interface هنا، حيث نستدعي الدوال بها بدون الحاجة لمعرفة ال Implementation (أي لا يوجد أي Reference لأي Concrete Types) ونعتمد على ال Factory في إنشاء هذا النوع الذي يطبق ال Interface

لنعرض كلاس ال RepositoryFactory (وهو يعتبر Compile Time Factory):

```
1.  public static IPersonRepository GetRepository(string type)
2.  {
3.      IPersonRepository repo = null;
4.
5.      switch (type)
6.      {
7.      case "Service":
8.          repo = new ServiceRepository();
9.          break;
10.
11.     case "SQL":
12.         repo = new SQLRepository();
13.         break;
14.
15.     case "CSV":
16.         repo = new CSVRepository();
17.         break;
18.
19.     default:
20.         throw new ArgumentException("Invalid Repository Type");
21.     }
22.
23.     return repo;
24. }
```

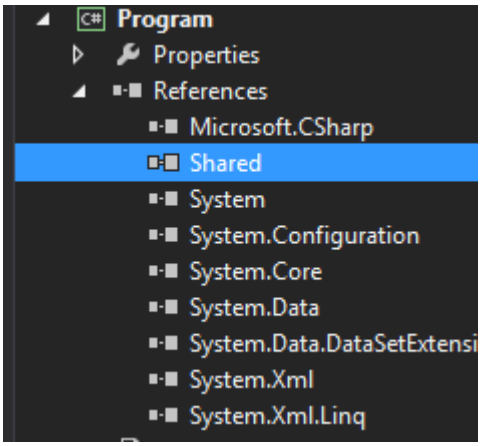
هناك بعض العيوب في هذا الكود أعلاه:

1) أنه يوجد لدينا Compile Time References للـ Implementation المختلفة، وهذا يعني أنه يتطلب وجود الـ Assembly References داخل Program Project واستخدام الـ Using لهذه الـ Concrete Classes حتى يمكن يعمل الكود. فماذا إذا أردنا أن ندخل نوع جديد من الـ Repository هنا مثلاً اضافه NoSqlRepository فسوف نحتاج إلى إضافة الأسمبلي (بعد عمل المشروع له) وعمل using له ومن ثم تحديث الكود لكي نضيف عمل ذلك النوع
2) أن الـ Factory يحتاج لـparameter لتحديد نوع الـ Repository، والذي ينادي الدالة يجب أن يقوم بتلك المهمة والأفضل هو أن نجعل مهمه تحديد النوع في مكان آخر مثلاً configuration file أو كلاس خاص بتحديد نوع الـ repo طالما سوف نحتاج نوع واحد فقط.
3) أنه في الغالب الـ Client قد يحتاج لنوع واحد فقط، وفي هذا الكود فكل هذه الـ Assembly يجب أن تكون متوفرة وقت الترجمة حتى يعمل الكود.

وسوف نحل هذه المشاكل بدلاً من استخدام Factory تستخدم Compile Time Binding إلى دالة تستخدم Runtime Binding أو Dynamic Loading وسوف نسميه الـ Dynamic Binding/Loading

لنبدأ العمل:

سنقوم أولاً بمسح الـ Assemblies للمكتبات الخاصة بالـ Repository في المشروع Program لأنه سيتم تحميل ما نريد وقت التشغيل (مع مسح الـ Using لها من داخل الكود وعمل الترجمة والتأكد من كل شيء على ما يرام).



سوف تتغير دالة الـ GetRepository وتصبح بهذا الشكل حيث تم تحميل الـ Assembly عن طريق الـ Reflection:

```
1.  public static IPersonRepository GetRepository()
2.  {
3.      string typeName = ConfigurationManager.AppSettings["RepositoryType"]; // (1)
4.      Type type = Type.GetType(typeName); // (2)
5.      object instance = Activator.CreateInstance(type); // (3)
6.      IPersonRepository repo = instance as IPersonRepository;
7.      return repo;
8.  }
```

شرح سريع للكود اعلاه:

1. جلب ال Assembly وال Type من ال Configuration
2. تحميل ال Assembly باستخدام ال Reflection
3. استخدام Activator Class لعمل ال Repository من ذلك النوع

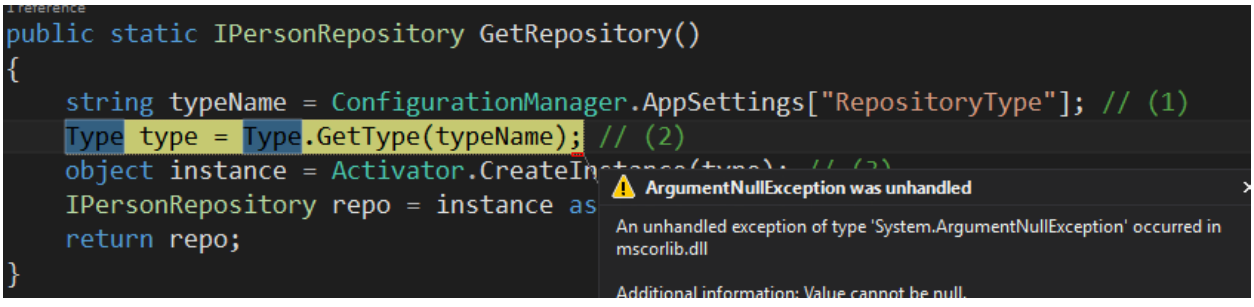
الآن سيصبح الكود في الدالة الرئيسية بهذا الشكل بعد عمل ال dynamic loading، ولاحظ عدم وجود أي معامل مرسل للدالة GetRepository

```
class Program
{
    static void Main(string[] args)
    {
        FetchData();
        Console.ReadKey();
    }

    private static void FetchData()
    {
        IPersonRepository repo = RepositoryFactory.GetRepository();
        IEnumerable<Person> people = repo.GetPeople();
        displayPeople(people);
    }

    private static void displayPeople(IEnumerable<Person> people)
    {
        foreach (var person in people)
        {
            Console.WriteLine(person.Id + " " +
                  person.FirstName + " " + person.LastName);
        }
    }
}
```

بالطبع في حال قمت بتشغيله الآن سوف تحصل على خطأ Exception والسبب أنه لم يتم تحديد نوع ال Repository في ال Configuration File



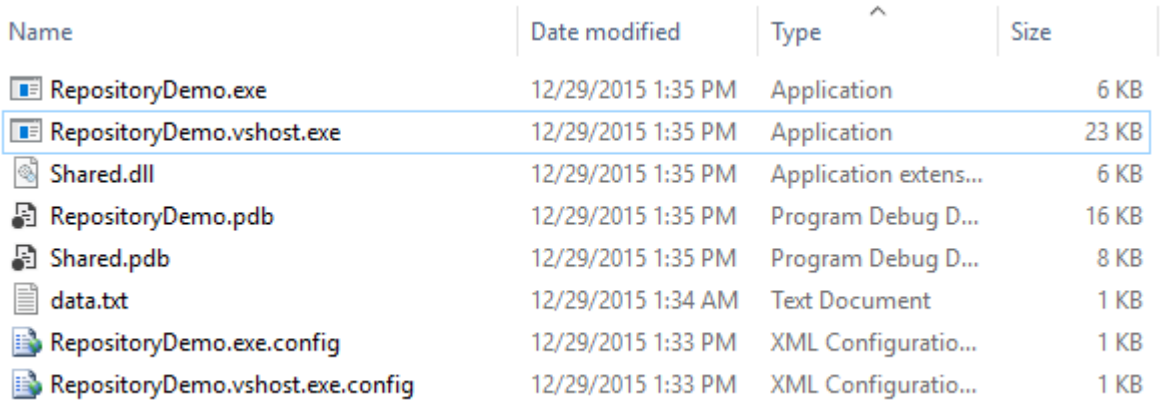
سوف نقوم الآن بفتح ملف ال configuration (ملف ال App.config) لتحديد ال Implementation وسنقوم بتحديد اننا نريد تحميل ال SQL Repository كما يلي:

```
<appSettings>
    <add key="RepositoryType"
            value="SQLDatabaseRepository.SQLRepository,
          ServiceRepository, Versio=1.0.0.0, Culture=neutral"/>
</appSettings>
```

وهي مسار ال Assembly حتى يمكن تحميله، ولمعرفة ال Full typed assembly name لأي كلاس لديك، فهناك عدة طرق اما باستخدام أدوات أو حتى بكتابة الكود التالي وتمرير النوع الذي تريد معرفته:

```
1.  Type type = typeof(Person)
2.  string fullname = type.AssemblyQualifiedName;
3.  Console.WriteLine(fullname);
```

تبقت الخطوة الأخيرة، في حال قمت بعمل Build للمشروع سوف تجد مجلد المشروع (اما Debug or Release، على حسب طريقة الترجمة، ونحن نعمل على ال Debug في هذا الكتاب) بهذا الشكل:



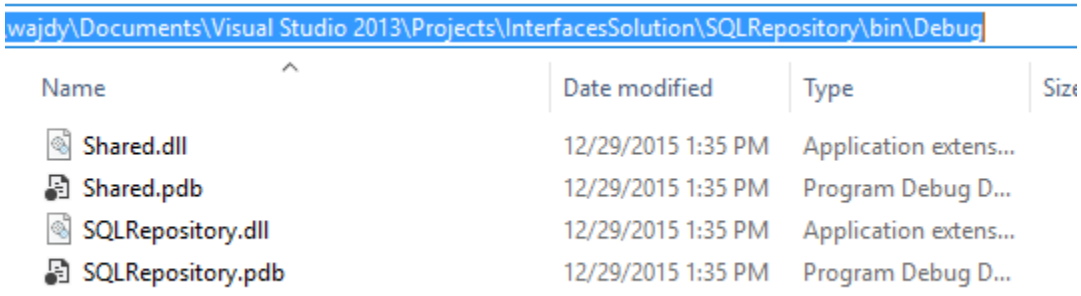
Name	Date modified	Type	Size
RepositoryDemo.exe	12/29/2015 1:35 PM	Application	6 KB
RepositoryDemo.vshost.exe	12/29/2015 1:35 PM	Application	23 KB
Shared.dll	12/29/2015 1:35 PM	Application extens...	6 KB
RepositoryDemo.pdb	12/29/2015 1:35 PM	Program Debug D...	16 KB
Shared.pdb	12/29/2015 1:35 PM	Program Debug D...	8 KB
data.txt	12/29/2015 1:34 AM	Text Document	1 KB
RepositoryDemo.exe.config	12/29/2015 1:33 PM	XML Configuratio...	1 KB
RepositoryDemo.vshost.exe.config	12/29/2015 1:33 PM	XML Configuratio...	1 KB

وطالما حددت ال Implementation الذي تريده في ملف ال Configuration فيجب أن يتواجد ال Assembly له في هذا المجلد وإلا فلن يتم إيجاده وستحصل على Exception السابق.

هناك عدة حلول منها تهيئة ال Visual Studio بحيث عندما يتم عمل Build لكامل ال Solution أو لأي Library Project فسوف يقوم مباشرة بنسخ الDLLs ونقلها مباشرة للمجلد الذي تريده وهكذا إذا طبقنا هذا الخيار على جميع ال Class Library سوف تجد أن المخرج اعلاه سوف يحتوي على كل ال DLLs وسيتم تحميل فقط ما تم اختياره في ملف ال Configuration.

لن نأخذ هذا المسار، وسوف نقوم بالعملية اليدوية وسننسخ ال SQL Repository DLL (الذي حددناه في ال Configuration) إلى هذا ال Folder، وسوف تجدها في المسار الخاص بال Project:

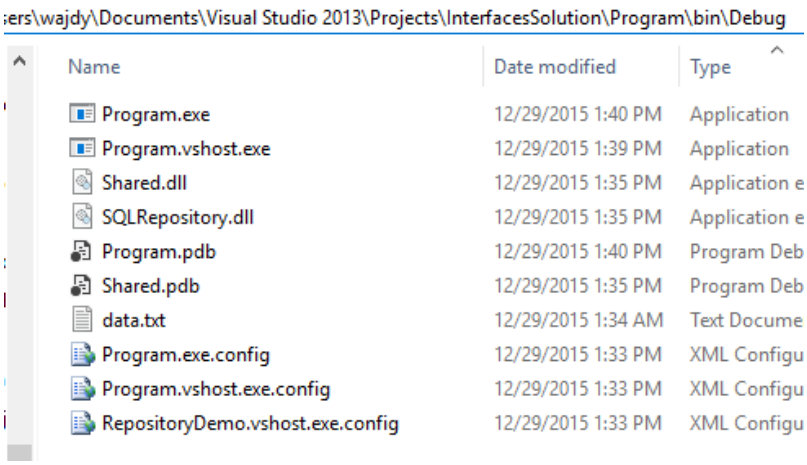
- قم بنسخ ال DLL



wajdy\Documents\Visual Studio 2013\Projects\InterfacesSolution\SQLRepository\bin\Debug

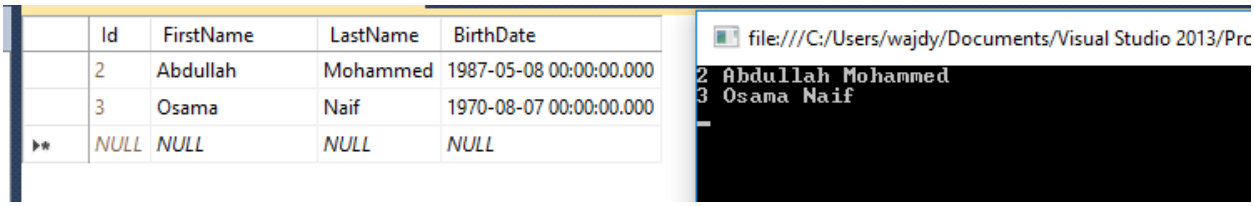
Name	Date modified	Type	Size
Shared.dll	12/29/2015 1:35 PM	Application extens...	
Shared.pdb	12/29/2015 1:35 PM	Program Debug D...	
SQLRepository.dll	12/29/2015 1:35 PM	Application extens...	
SQLRepository.pdb	12/29/2015 1:35 PM	Program Debug D...	

- وضعها في مجلد ال Debug لل Program:



كل شيء الآن جاهز، الكود سوف يقوم بعمل Loading للـ Assembly التي يوجد اسمها في ملف الـ Configuration، والـ Assembly موجودة.

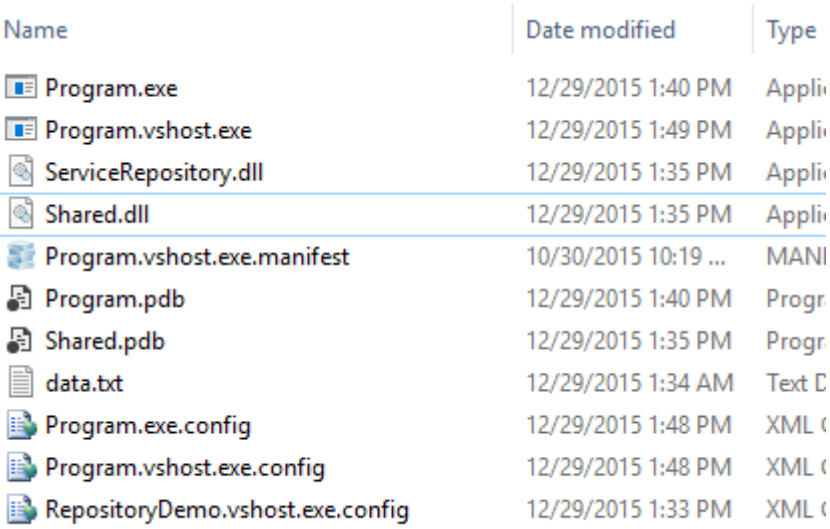
عندما تشغل المشروع سوف يتم تحميلها وسوف تحصل على البيانات من قاعدة البيانات صحيحاً، وهذا هو المخرج مع البيانات في قاعدة البيانات:



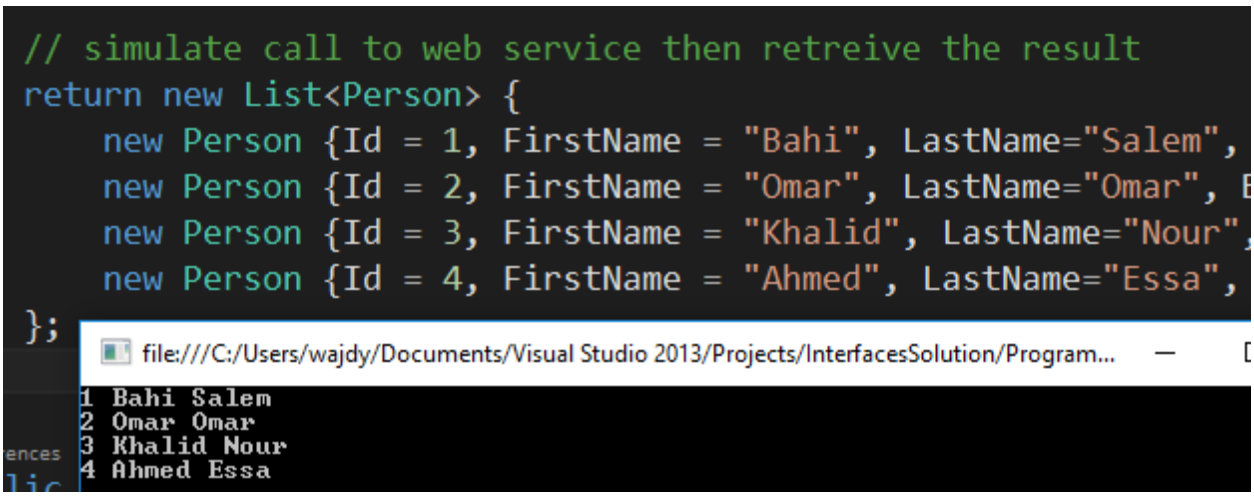
لنقم الآن بتغيير الـ Assembly Name من الـ Configuration ووضع الـ Service Repository بدالها، ونضعها ايضاً على مجلد الـ Program (في الـ Debug إذا كنت تشغل مشروعك بهذا الطور أو الـ Release إذا كنت تعمل بهذا الطور)، وهكذا سوف يكون شكل الـ Configuration:

```
1.  <appSettings>
2.      <add key="RepositoryType" value="WebServiceRepository.ServiceRepository,
3.                       ServiceRepository, Versio=1.0.0.0, Culture=neutral"/>
4.  </appSettings>
```

بالطبع لا تنسى نسخ الـ DLL Assembly لمجلد المشروع (طبعاً تستطيع حذف أي Assembly لا تستخدم):



هكذا عندما يعمل البرنامج سوف تلاحظ أنها نفس البيانات التي تأتي من ال Web Service Repository، ولاحظ أنه تم تغيير ال Repository بدون إعادة الترجمة، فقط بتغيير ال configuration وهذه هي الفكرة من هذا الفصل، **ممتاز**.



أخيراً إذا أردت اضافه Repository جديدة فقد تقوم بإضافتها في المجلد ومن ثم تعديل ال configuration بدون تغيير أي جزئية في الكود، ايضاً الميزة الأخرى انه إذا كان العميل فقط يريد مثلا SQL Repository فلا توجد حاجة بوضع تلك ال assembly في المجلد ووضع ما سوف يستخدم فقط

هكذا لدينا **Complete Decoupled Application** عن طريق الاعتماد على Programming to Interface ومن ثم Dynamic Loading Types.

الفرق بين ال Compile Time Factory وال Dynamic Factory

- **في المعاملات Parameters:**

- في ال Compile Time Factory تحتاج لمعامل يقوم بتحديده من يستدعي ال Factory
- بينما في ال Dynamic Factory فهي لا تحتاج لأي معامل لكي يتم تحديد النوع، بل الدالة سوف تستخدم ال Configuration file لكي تحدد نوع ال Implementation المطلوب وتقوم بإنشائه

- **في الربط Binding:**
  - في ال Compile Time لكي يتم ال Binding فيجب أن تكون ال Assemblies وال References لل Implementations موجودة حتى يترجم صحيحاً.
  - بينما في ال Run-Time فلا تحتاج لأي References/Assemblies اثناء الترجمة، وسيتم تحميل ال assembly وقت التشغيل run time (باستخدام ال Reflection).

خلاصة

- تم شرح في هذا الفصل طريقة ال Dynamic Binding وكيف يمكن تحميل الكلاسات عن طريق ال Reflection وجعل اسم الRepository في ال Configuration File.
- توضيح الفرق بين ال Compile Time Factory وال Dynamic Binding

الفصل الخامس: مقدمة للطبقات وفصل الاهتمامات Application Layering

في هذا الفصل سوف نأخذ مثال عملي حول مشروع صغير وكيف يمكن برمجته بالطريقة العادية من خلال الدوال، وكيف يمكن حلها بطريقة تستخدم ال Layered Architecture. وسوف يكون الفصل عملي بحت ولن نتحدث نظرياً عن الطبقات وال Patterns المستخدمة في كل منها والفروق بينهم حيث هذه مهمة أحد أجزاء السلسلة. الهدف هنا فقط معرفة كيف يمكن أن تستخدم ال Interface في تصميمات مهندسة بطريقة أفضل.

معمارية البرنامج Application Architecture

لا يمكنك بناء مشروع قابل للصيانة Maintainable أو قابل لتوسعة Scalable بأساسيات ضعيفة، والتخطيط لمعمارية تستحمل طلبات العميل في الوقت الحالي والمستقبل القريب أمر مهم للنجاح، وهذا الأمر يفرق بين المبرمج عادي ومبرمج آخر يستطيع تصميم ال Design للمشروع ورسم الخطوط العريضة بين الوحدات والتداخل فيما بينها.

مثال عرض المنتجات من قاعدة البيانات

لنفرض أنك تعمل على صفحة لعرض قائمة المنتجات Products وكل منتج يتكون من رقم المنتج، اسم المنتج، سعر التجزئة، سعر البيع. ونريد أن تقوم بعرض المنتجات في الصفحة بالإضافة إلى عرض التخفيض في كل منتج وكم نسبة التخفيض. وتستطيع معرفة التخفيض من الفرق بين سعر التجزئة وسعر البيع. ال recommended retail price تعني سعر التجزئة الموصي به وسوف نستخدم الإختصار RRP.

	Id	Name	RRP	SellingPrice
	1	Java Book	109.5000	108.0000
	2	Android Mobile	500.0000	510.0000
	3	Router	300.0000	300.0000
▶*	NULL	NULL	NULL	NULL

في حال كان هذا المشروع موقع ويب او Desktop أو حتى تطبيق موبايل (يعرض البيانات في UI) فعادة ما تجد المبرمج يكتب كود متداخل (كود التعامل مع القاعدة داخل كود ال UI) بحيث يصعب استخدام أي جزئية في صفحة ثانية مثلاً في صفحة ال Admin لعرض المنتجات. وكثير ما يتم نسخ الكود كما هو وبالتالي أي تغيير بسيط سوف تحتاج التغيير في أماكن كثيرة لهذا السبب.

لتبسيط المثال سوف نكمل على طريقة ال Console Application (حتى يكمل معنا مبرمجي اللغات الاخرى) ولنرى أول طريقة لحل المشروع، وكل ما نريده هو طباعة قائمة المنتجات بالإضافة إلى عرض التخفيض ونسبته في كل منتج.

بالطبع طريقة التعامل مع قاعدة البيانات خارج نطاق الموضوع، وهنا استخدمنا LINQ To SQL وقمنا بتوليد ال Database Context لكن لا تقلق نفسك بها (إذا لم تكن ملم بها)، كل ما يهم الآن لدينا دالة اسمها GetProducts ترجع قائمة بالمنتجات بغض النظر عن الكيفية (سواءً كانت مجرد استعلامات مباشرة بالقاعدة أو باستخدام أي من ال ORMs المتوفرة).

```
static void Main(string[] args)
{
    var products = GetProducts();

    foreach (var product in products)
    {
        string discount = DisplayDiscount(product.RRP, product.SellingPrice);
        string saving = DisplaySavings(product.RRP, product.SellingPrice);

        Console.WriteLine(String.Format("Name: {0} \nPrice: {1:C} \nRRP: {2:C} " +
              "\nDiscount: {3} \nSaving: {4}\n",
              product.Name, product.SellingPrice, product.RRP, discount, saving));
    }

    Console.ReadKey();
}
```

لكل منتج سوف نقوم باستدعاء دالة لعرض التخفيض، ودالة لعرض نسبة التخفيض كما يلي:

```
private static string DisplayDiscount(decimal RRP, decimal SalePrice)
{
    string discountText = "";
    if (RRP > SalePrice)
        discountText = string.Format("{0:C}", (RRP - SalePrice));
    return discountText;
}

private static string DisplaySavings(decimal RRP, decimal SalePrice)
{
    string savingText = "";
    if (RRP > SalePrice)
        savingText = (1 - (SalePrice / RRP)).ToString("#%");
    return savingText;
}
```

الآن إذا قمت بتشغيل البرنامج فسوف ترى المخرج يعمل بشكل صحيح:

لنفرض أن العميل قام بإضافة طلب جديد new business requirement وهو اضافة Dropdown List يختار منها نوع العميل (هل هو عميل عادي، أم تاجر) وعلى اساس ذلك يتم عمل تخفيض (العميل العادي ليس له تخفيض، بينما للتاجر لديه تخفيض بنسبة معينة).

وتم ابلاغك ايضاً قد تكون هناك أنواع اخرى من التخفيضات بناء على اشياء اخرى بالمستقبل. لأننا نعمل على Console (ولا وجود ل Dropdown List) سوف نسأل المستخدم عنه نوعه مثلاً 1 للتاجر و 2 للمستخدم العادي وعلى اساسه سوف نقوم بعرض التخفيض.

قد تقوم بهذا الحل، وسوف تسأل المستخدم اولاً عن نوعه ومن ثم تستدعي دالة الخصم (سطر 6):

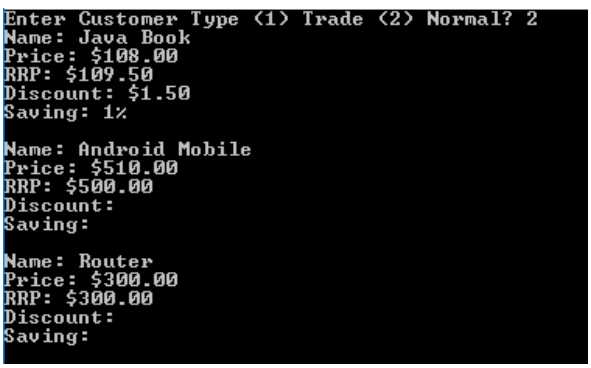
```
1.  static void Main(string[] args)
2.  {
3.      var products = GetProducts();
4.
5.      Console.Write("Enter Customer Type: ");
6.      int customerType = int.Parse(Console.ReadLine());
7.
8.      foreach (var product in products)
9.      {
10.         decimal newSellingPrice = ApplyDiscount(product.SellingPrice, customerType);
11.
12.         string discount = DisplayDiscount(product.RRP, newSellingPrice);
13.         string saving = DisplaySavings(product.RRP, newSellingPrice);
14.
15.         Console.WriteLine(String.Format("Name: {0} \nPrice: {1:C} \nRRP: {2:C} " +
16.             "\nDiscount: {3} \nSaving: {4}\n",
17.              product.Name, newSellingPrice, product.RRP, discount, saving));
18.     }
19.
20.     Console.ReadKey();
21. }
```

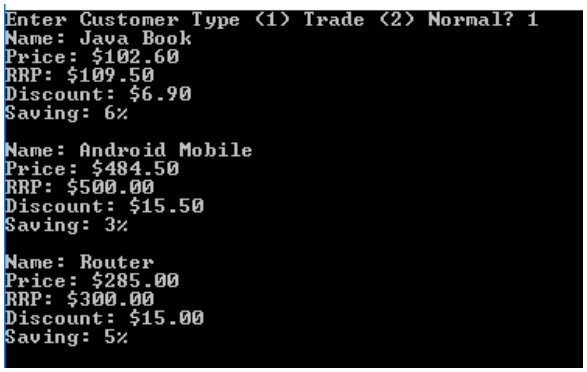
لاحظ أن السعر الجديد هو الذي سيحسب في الخصم ونسبة التوفير.

دالة الخصم كالتالي:

```
1.  private static decimal ApplyDiscount(decimal originalPrice, int customerType)
2.  {
3.      decimal price = originalPrice;
4.
5.      if (customerType == 1)
6.      {
7.          price = price * 0.95M;
8.      }
9.
10.     return price;
11. }
```

بهذا الشكل سوف يكون المخرج (بعد تشغيله مرتين مرة بالقيمة 1والثانية بالقيمة 2):





بهذا الشكل لو طلب منك أي اضافة سوف تقوم بتغيير دالة ال Apply Discount وعلى اساسها سوف تقوم بعملية الخصم.

المبرمج المبتدئ قد يضع كل الكود داخل ال Main بدون أي دوال مساعدة، وبالطبع هذا تصميم خاطئ. وتقسيم المهام إلى دوال هذه تعتبر أول خطوة في العمل الصحيح (كما قمنا أعلاه).

المبرمج الأفضل قد يقوم بنقل هذه الدوال ال private static إلى كلاس Helper فيه هذه الدوال (وغيرها من الدوال المشتركة) حتى تستطيع استخدامها في مكان آخر إن احتجت لهذه المعادلات.

الطلب الجديد من العميل أن هذا المشروع يريده أن يعمل على Web Interface ايضاً بالإضافة إلى Console. بمعنى أن لديك مشروعين داخل ال Solution الأول لل Console والآخر لل Web. بالطبع المشروعين بحاجة لهذه الدوال المساعدة، وتستطيع نسخ الكلاس Helper من المشروع الأول إلى المشروع الثاني، ولكن مرة اخرى في حال حدث أي تغيير فسوف تحتاج تغييرهم الاثنين معاً.

الحل الأفضل هو أن تضع ال Helper في Class Library وتربطه مع مشروع الويب ومشروع الكونسول وهكذا الجميع يتشارك في ال Business. ويمكنك أن ترجع للملحق 1 للمزيد عن ال Class Library إذا لم تكن قد قرأته من قبل.

هذا الاسلوب سهل حقاً، أي مبرمج يستطيع العمل فيه والدخول مع فريقك اذا كنت تستخدم هذا النمط، وهذه الطريقة تتبع ل Pattern معروف اسمه Transaction Script والفكرة فيه أن كل الBusiness سوف يكون موجود على Static Methods تقوم بكل شيء (تأخذ طلب المستخدم، تقوم بعمل ال Validation، تفحص القيم وتقوم بالعمل على أساسها، تقوم بالعمل على القاعدة وتخزين البيانات، تقوم بعمل Logging)، بالطبع لا تنسى أن كل هذه الأعمال سوف تكون مقسمه لدوال اخرى (وتكون الدالة الأولى التي يتعامل معها العميل هي التي تنادي كل الدوال الصغيرة لإتمام العمل).

ال Transaction Script Pattern هو أحد ال Patterns المناسبة للمشاريع الصغيرة والتي لا تتغير متطلباتها كثيراً، لكن كل ما يكبر نطاق العمل سوف يعقد المشروع ويصعب العمل عليه فيما بعد، فأنت هنا حقيقة تبرمج بطريقة اجرائية Procedures ولا تستفيد من ال Object Oriented حيث كل شيء لديك في دوال، وهذا ليس عيباً ولكن هناك طرق اخرى أفضل وأكثر مناسبة لطبيعة البرمجيات المؤسساتية Enterprise Applications التي بها أكثر من شخص في الفريق، والتي تتغير متطلباتها دائماً وتكبر وتزداد المهام بها مع الوقت.

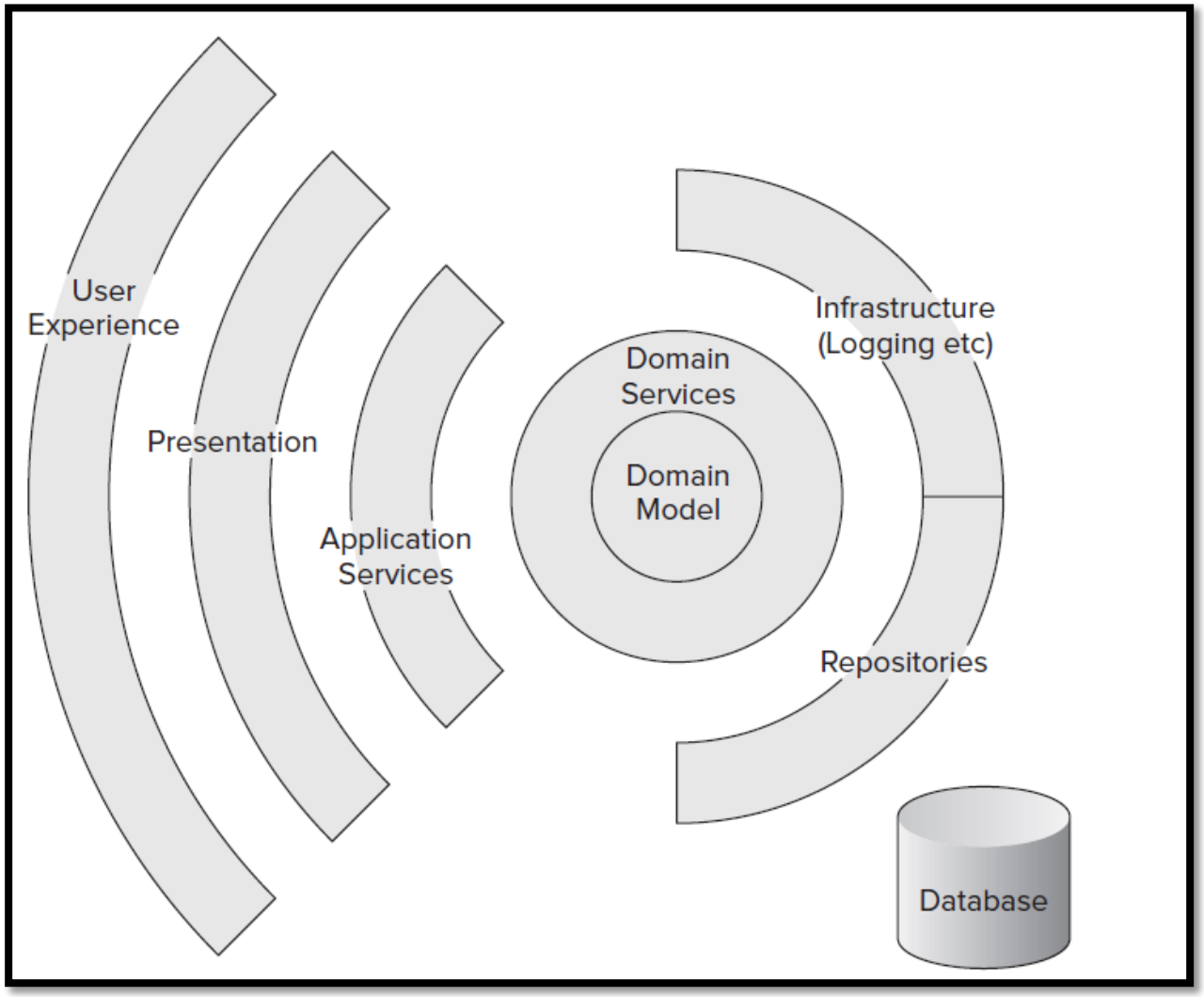
فيما يلي سوف نقوم بنفس المشروع ولكن سوف نتبع Pattern آخر اسمه Domain Model ولن نسهب في شرحه لأن هناك جزء مخصص لهذه الأمور ولكن سوف نقوم بالجزء العملي مباشرة. ولكن سوف نتحدث اولاً عن فصل الاهتمامات.

فصل الاهتمامات Separating your Concerns

من أهم الطرق لترتيب المشروع هو فصل كل الأمور إلى اجزاء، سواء كانت على مستوى مجلدات داخل المشروع، أو حتى لها Namespace مختلفة أو حتى تكون Projects منفصلة من نوع Class Library.

التطبيق السابق للمشكلة كان الكود كله موجود داخل ملف واحد فقط، ولكن الآن سوف تختلف الصورة تماماً، عندما ننتهي سوف نشغل البرنامج وسوف ترى نفس المخرج السابق ولكن الفرق سوف يكون في طريقة التصميم وافضليتها وأنها أكثر مرونة Adaptive وقابلة للعمل وإضافة المزيد بعكس الاولى التي يصعب العمل عليها فيما بعد خصوصاً عندما يكبر المشروع وتزداد المهام.

سوف نطبق المعمارية التالية (وغالباً الذي يبني المعمارية لديه هدف معين من كل طبقة، وأحياناً قد لا تحتاج لكل هذه الطبقات وتكتفي ب 2 او 3).

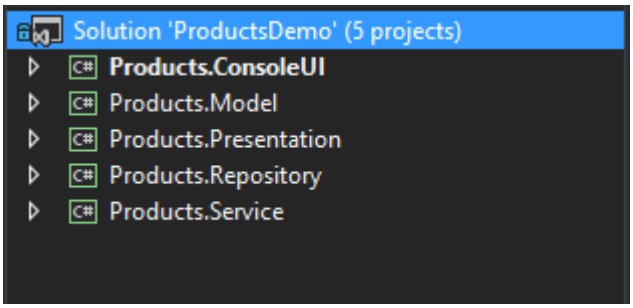


الصورة قد تبدوا غريبة ولكن سوف يزداد الأمر وضوحاً عندما ننهي برمجة كل Layer. ولكن حتى نعطيك فكرة بسيطة (سوف تتضح هذه النقاط عندما تبدأ التطبيق العملي):

- طبقة ال User Experience وهي طبقة الواجهة UI التي يعمل عليها المستخدم سواء كانت Web UI أو Console UI أو حتى Desktop/WinForm Application.
- طبقة ال Presentation وهي طبقة قد لا تحتاجها في بعض المشاريع، ولكن الفكرة هنا أننا نريد أن نطبق الMVP Model View Presenter واختصارا وهو أحد ال Patterns الذي يسهل لنا العمل والاختبار ايضاً، وسوف ترى فيما بعد أننا أنشأنا مشروع Web UI وConsoleUI بكل سهولة بقليل من الكود بفضل هذه الطبقة
- ال Service Layer وهي الطبقة الاولى أو ال Entry Point للطلبات، وسوف تأخذ القيم المدخلة من المستخدم Request وترجع لك ال Response المناسب (سوف نتبع ال Request-Response Messaging Pattern هنا) ايضاً لن ترجع لك هذه الطبقة المنتج من قاعدة البيانات كما هو بل ستقوم بتحويل المنتج المخزن في قاعدة البيانات Database Model إلى منتج مناسب لكي يتم عرضه للمستخدم View Model وقد يحتوي على متغيرات أكثر حتى تعرض للمستخدم.
- الطبقة الرابعة ال Domain/Business Layer وهي بها كل ال Business في التطبيق (عملية الخصم بناء على نوع العميل) وتطبيق الخصم والنسبة. سوف تتعامل ال Server Layer مع هذه الطبقة Domain/Business وهي بدورها سوف تتعامل مع ال Repository. التعامل هنا سوف يكون Interface وبالتالي ال Business Layer سوف تعمل بغض النظر عن نوع ال Repository سواء كان SQL أو كان ملف أو أي نوع آخر. سوف تعرف خلال هذه الجزئية أن إنشاء نوع ال Repository لن يكون من مسؤولية ال Business وانما هي ستتعامل مع ال Injected Object بغض النظر عن نوعه. (هذا يعرف ب Dependency Injection ولن نشرح هذا الموضوع وانما نكتفي بالإشارة له حيث نخصص له جزء منفصل في السلسلة أن شاء الله).
- الطبقة الاخيرة Repository Layer وهي التي تتعامل مع الData Source
- هناك طبقة اخرى مختصة ببعض الأمور التي تحتاجها في كل الطبقات مثلاً ال Validations أو ال Logging، مثل هذه الطبقات يمكن أن تستدعي الدوال التي بها ك Helper Methods أو الافضل أن تتبع مبدأ ال Aspect Oriented Programming بها لكن ايضاً هناك جزء مخصص لهذا الأمر.

ولنبدأ من الصفر منذ عمل مشروع جديد. ويمكنك أن تعمل على المشروع السابق أو تنشي مشروع جديد لو أردت.

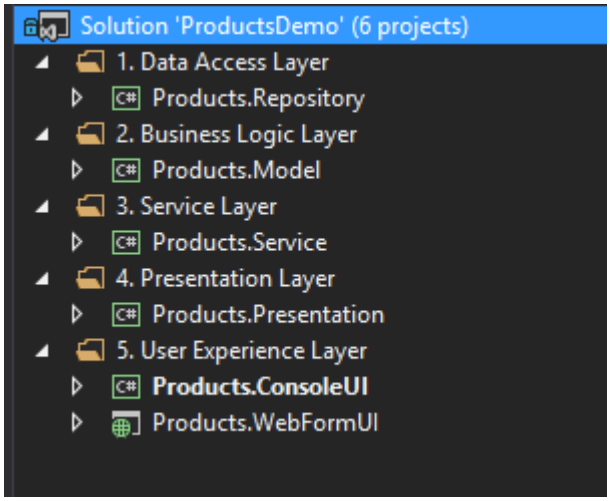
بعد ذلك أضف 4 انواع من ال Class Library Projects (بالإضافة إلى Console App) بالأسماء التالية:



نأتي للربط (عن طريق اضافة ال Reference) فقم بإضافة:

- ال Repository سوف تستخدم ال Model
- ال Service سوف تستخدم ال Model وال Repository
- ال Presentation سوف تستخدم ال Model وال Service
- ال ConsoleUI سوف يستخدمهم جميعهم

يمكنك وضعهم داخل مجلدات (عن طريق اضافة مجلد ومن ثم سحب Drag and Drop كل مشروع لداخل المجلد المناسب) بحيث يكون مشابه للصورة التالية وهذا يعطي ترتيب أفضل ويسهل للاستخدام:



سنبدأ بالبرمجة وسنبدأ بأهم طبقة في المشروع ألا وهي ال Business Layer (أحياناً ترى أسماء مثل Business Logic Layer أو حتى بالاختصار BLL).

طبقة ال Business Layer

طالما نحن نتبع الاسلوب Domain Model Pattern فسنقوم بعمل domain model للمشكلة التي نريد حلها ويمكنك التفكير بأنها Conceptual Model تحتوي على كل ال Entities الموجودة في النظام مع العلاقات فيما بينها. وال Domain Model لا يشترط أن يكون مساوياً لل Table Model الذي يوجد على القاعدة، بمعنى لا يجب أن يكون هناك One-to-One Mapping لكل الحقول ولكن ال Domain Model يجب أن يكون Rich بحيث يمثل كل شيء بلغة أسهل لل Business.

لتتذكر المشكلة التي نعمل عليها، في كانت لعرض المنتجات Products وعرض الأسعار Price لها بعد تطبيق الخصومات التي تختلف بناء على عوامل مختلفة. لذلك منطقياً في المشكلة هذه لدينا Product وPrice سوف يكونا ال Domain Models هنا. وكل ال domain models التي سوف ننشئها سوف تكون في مشروع ال Model.

بالنسبة للخصومات فطالما هي تتغير وسوف تتغير في المستقبل، فهذا مكان مناسب لتطبيق ال Abstraction وسوف نقوم باستخدام ال Interface هنا في هذا الأمر.

لذلك سنقوم بعمل Interface اسمه IDiscountStratgey يمثل طريقة الخصم بالطبع بدون Implementation واسمه وسوف نقوم بعمل عدة Implementations كل منها بطريقة خصم معينة.

سبب التسمية DiscountStrategy يعود لأننا استخدمنا ال Strategy Pattern وهي مجموعة من الخوارزميات التي يمكن أن تستخدمها وتبدلها وقت التشغيل.

الكود التالي يعرض الInterface :

```
1.  namespace Products.Model
2.  {
3.      public interface IDiscountStrategy
4.      {
```

```
        decimal ApplyExtraDiscountTo(decimal originalSalePrice);
    }
}
```

بعد ذلك نعرض ال Implementations وهي مرة تكون بخصم للتاجر TradeDiscountStrategy

```
namespace Products.Model
{
    public class TradeDiscountStrategy : IDiscountStrategy
    {
        public decimal ApplyExtraDiscountTo(decimal originalSalePrice)
        {
            decimal price = originalSalePrice;
            price = price * 0.95M;
            return price;
        }
    }
}
```

وهذا ال Implementation للعميل الذي لا خصم عليه

```
namespace Products.Model
{
    public class NullDiscountStrategy: IDiscountStrategy
    {
        public decimal ApplyExtraDiscountTo(decimal originalSalePrice)
        {
            return originalSalePrice;
        }
    }
}
```

وطالما لا خصم عليه فسوف نرجع السعر كما هو، ولاحظ أسم الكلاس يبدأ ب Null وهو ايضاً سمي بسبب اتباعنا ل Pattern اسمه Null Design Pattern وهو ببسطة الكلاس الذي لا يقوم بأي وظيفة، وانا يطبق كل ال methods بحيث لا يكون هناك أي عمل، وطالما لا يوجد خصم فسوف يرجع السعر كما هو.

سوف نقوم الآن بعمل كلاس يمثل السعر Price وهو الذي سوف يطبق الخصم على حسب نوعية العميل ونسبة التوفير وكل شيء يتعامل مع سعر المنتج.

وفيما يلي كود ال Price:

```
namespace Products.Model
{
    public class Price
    {
        private IDiscountStrategy _discountStrategy = new NullDiscountStrategy();
        private decimal _rrp;
        private decimal _sellingPrice;

        public Price(decimal RRP, decimal SellingPrice)
        {
            _rrp = RRP;
            _sellingPrice = SellingPrice;
        }

        public void SetDiscountStrategyTo(IDiscountStrategy DiscountStrategy)
```

```
16.         {
17.             _discountStrategy = DiscountStrategy;
18.         }
19.
20.         public decimal SellingPrice
21.         {
22.             get { return _discountStrategy.ApplyExtraDiscountTo(_sellingPrice); }
23.         }
24.
25.         public decimal RRP
26.         {
27.             get { return _rrp; }
28.         }
29.
30.         public decimal Discount
31.         {
32.             get
33.             {
34.                 if (RRP > SellingPrice)
35.                 return (RRP - SellingPrice);
36.                 else
37.                 return 0;
38.             }
39.         }
40.
41.         public decimal Savings
42.         {
43.             get
44.             {
45.                 if (RRP > SellingPrice)
46.                 return 1 - (SellingPrice / RRP);
47.                 else
48.                 return 0;
49.             }
50.         }
51.     }
52. }
```

يستقبل هذا الكلاس السعر وسعر التجزئة في دالة البناء، وهناك دالة تستقبل ال IDiscountStrategy (بالطبع سوف يرسل أحد ال Implementation لها) وعلى اساسه عندما يتم استدعاء ال Selling Price في سطر 22 سوف يتم تطبيق ذلك ال Implementation على السعر.

بعد ذلك ما تبقى مجرد استرجاع لل RRP، وفيه دوال ال Discount وال Saving السابقة ولكن حالياً بدلاً من جعلها دوال static تم وضعها داخل ال Price وجعلها Property فيه.

ملاحظات:

- سطر 20 كان بالامكان عمله عن طريق دالة عادي اسمها مثلاً GetSellingPrice ولكن تم استخدام خاصية Property في السي#. المهم انه سوف يتم استدعاء الخصم بغض النظر عن نوعه سواء كان Trade أو Null.
- اسلوب ارسال الكائن للدالة SetDiscountStrategy يعتبر نوع من أنواع ال Dependency Injection ألا وهو Setter Injection والفكرة أنك ترسل ال Implementation للدالة بدلاً من جعلها هي التي تقوم بإنشائه.

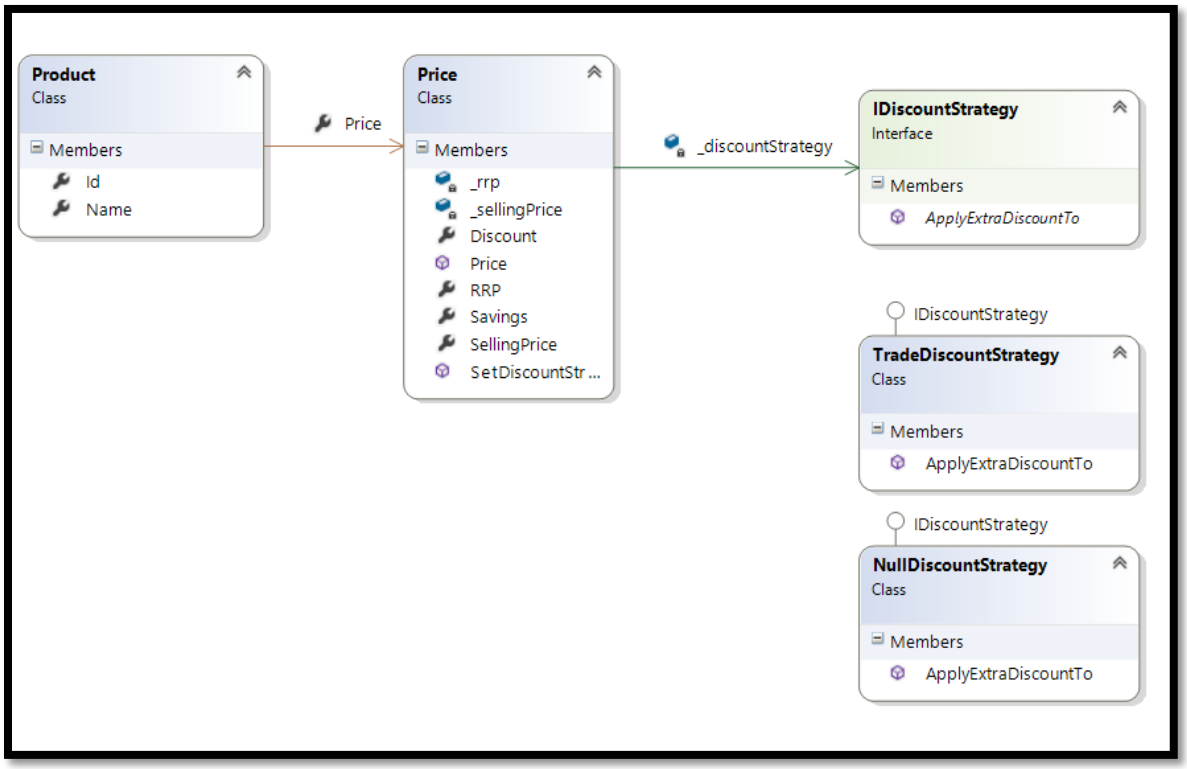
الآن سوف ننشئ كلاس يمثل ال Product وفيه الإسم Name وال Id وال Price. لأن ال Price بها كل شيء (السعر، سعر التجزئة، طريقة الخصومات).

```
1.  namespace Products.Model
2.  {
3.      public class Product
4.      {
5.          public int Id { get; set; }
6.          public string Name { get; set; }
7.          public Price Price { get; set; }
8.      }
9.  }
```

مرحباً بك في عالم ال Object Oriented الحقيقي!

الكثير من المبتدئين من يعتقد أن ال Object Oriented هي فقط في لتمثيل الكائنات، مثلاً ال Employee وال Animal أو كلاس Product ويقوم بعمل مجموعه من دوال ال Set/Get بها. لكن هذا جانب واحد منها، والفائدة الأكبر هي في الكائنات التي تقوم بعمل حقيقي Business مثلاً EBook Parser أو TaskSchedular أو PaymentProcessor أو حتى كلاس Price أعلاه. فعندما تستخدم أي لغة من لغات ال Object Oriented وتقوم بعمل كائنات Concept Objects (تحتوي على Set/Get فقط) فهذا قد لا يكون كود Object Oriented ولكن الحقيقة هو استخدام الكائنات Objects ولكن ضمن سياق Procedural Logic.

لنشاهد الآن ال Class Diagram لما قمنا بعمله:



بدلاً من التعامل مع الأرقام لتحديد نوع المستخدم، سوف نقوم بعمل Enumeration اسمه Customer Type يدل على نوع المستخدم ودائماً لا تستخدم أي أرقام داخل المشروع واستخدم الثوابت بدلاً منها:

```
namespace Products.Model
{
    public enum CustomerType
    {
        Standard = 0,
        Trade = 1
    }
}
```

سوف نضيف الآن ال DiscountFactory والتي ترجع نوع ال Discount Implementation على حسب نوع العميل.

```
namespace Products.Model
{
    public class DiscountFactory
    {
        public static IDiscountStrategy GetDiscountStrategyFor(CustomerType type)
        {
            switch (type)
            {
            case CustomerType.Trade:
                return new TradeDiscountStrategy();
            default:
                return new NullDiscountStrategy();
            }
        }
    }
}
```

تبقى بضعه كلاسات علينا أن نضيفها، ويمكن أن نجعل ال Service Layer تتعامل مباشرة مع الكلاس DiscountFactory وتقوم بالعمليات كلها، لكن لكي نجعل لكل طبقة واجهة Service مسؤولة عن كل العمليات التي بداخل الطبقة (هذا في ال Business/Model وال Service) وهذا ينظم العمل بطريقة أفضل. سوف يتم استخدام الاسلوب Layer.Serivce لكي تعرف عن أي Service class نتحدث (لأنه سوف يكون هناك كلاسين بالاسم Service، الأول هو في طبقتنا الحالية والأخر في طبقة ال Service Layer).

ال Model.Service سوف يتعامل مع ال Repository Layer والتعامل سوف يكون عن طريق Interface لذلك سوف يتم وضع ذلك ال Interface هنا

```
namespace Products.Model
{
    public interface IProductRepository
    {
        IList<Product> FindAll();
    }
}
```

أيضاً ال Model.Service تحتاج دالة تحصل من خلالها على كل المنتجات وطريقة الخصم وبعدها يتم تطبيق الخصم على كل المنتجات، سوف نضع هذه الدالة هنا ويمكن أن تكون public static لكن نفضل أن تكون ك Extension method بحيث يتم استدعاء هذه الدالة كدالة داخل ال List. وهذه هي وظيفة الكلاس التالي:

```
1.  namespace Products.Model
2.  {
3.      public static class ProductListExtensionMethods
4.      {
5.          public static void Apply(this IList<Product> products,
6.                  IDiscountStrategy discountStrategy)
7.          {
8.              foreach (Product p in products)
9.              {
10.                 p.Price.SetDiscountStrategyTo(discountStrategy);
11.             }
12.         }
13.     }
14. }
```

يمكن أن تعتبره أنه مجرد Helper Class يحتوي على دالة static عادية، لكن الفكرة سوف نستخدم خاصية في ال #C تسهل عملية استدعاء هذه الدالة كما ستشاهد فيما بعد (لكن تستطيع استبدالها بدالة عادية).

هذا هو كود ال Model.Service:

```
1.  namespace Products.Model
2.  {
3.      public class ProductService
4.      {
5.          private IProductRepository _productRepository;
6.
7.          public ProductService(IProductRepository productRepository)
8.          {
9.              _productRepository = productRepository;
10.         }
11.
12.         public IList<Product> GetAllProductsFor(CustomerType customerType)
13.         {
14.             IDiscountStrategy discountStrategy =
15.                     DiscountFactory.GetDiscountStrategyFor(customerType);
16.
17.             IList<Product> products = _productRepository.FindAll();
18.             products.Apply(discountStrategy);
19.             return products;
20.         }
21.     }
22. }
```

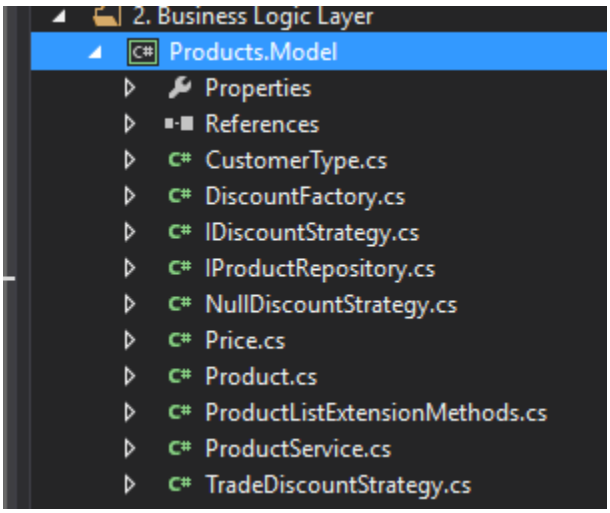
الكود أعلاه يعتبر قلب ال Model/Business Layer حيث من خلالها سوف تتعامل طبقة ال Service، وهو بسيط ولكن يقوم بتنظيم سير العمل وكل شيء (تذكر الطريقة التقليدية كانت بالدوال ولكن هنا أصبح باستخدام ال Object Oriented)، وهذا شرح بسيط للكلاس أعلاه:

1) دالة البناء تأخذ نوع ال Repository في دالة البناء بغض النظر عن ال Implementation وسوف يتم استدعاء كل المنتجات في سطر 17.
2) الدالة GetAllProductsFor تقوم بكل العمل فهي:
    1. ترجع كل المنتجات على حسب نوع العميل وذلك أولاً من خلال جلب ال Implementation المناسب لل Discount (في سطر 14-15)
    2. بعدها يتم تطبيق الخصم باستخدام ال Extension Method، ولاحظ فائدة ال Extension Method في سطر 18 حيث تم استدعاء دالة ال Apply على جميع

المنتجات كما لو كانت هي دالة داخل ال IList إذا لم تقم بعمل ال Extension Method فتستطيع استدعاء الدالة مباشرة من خلال ClassName.MethodName.

مرة أخرى الكود اعلاه يمثل واجهة ال Business/Model Layer حيث ستتعامل ال Service Layer مع ال Model.ProductService وهي ستقوم بالعمل الداخلي كما في دالة GetAllProductsFor.

بهذا الشكل تكون قد انتهيت من ال Business Layer وهي أكبر جزء في التطبيق، ولاحظ لم نتعامل مع أي Data Store معينة وسوف تعمل على أي طريقة تستخدم ال IProductRepository سواء كانت SQL أو ملف والخ. وهذه شكل ال Project بعد الانتهاء من هذه الطبقة



بالطبع يمكن عمل ترتيب ايضاً داخل ال Project نفسه، مثلاً اضافة مجلد اسمه Discounts ووضع كل ال Implementations فيه، لكن حالياً سوف نكتفي بالشكل اعلاه.

ال Service Layer

ال Service.Service Layer تعتبر أول مرحلة يتعامل بها التطبيق، وهي تعمل ك Facade حيث تقوم هذه الطبقة بإرجاع View Model (احياناً يسمى Presentation Model) وهو Model أكثر فائدة من ال Business Model حيث يقدم للواجهة كل ما تحتاجه من بيانات لتلك الصفحة. لذلك نقول هو More Optimized for Specific Views وأحياناً قد يحتوي ايضاً على دوال تساعد في تقديم كافة البيانات.

ال Façade Design Pattern

وهو Design Pattern فكرته أن يكون لديك كلاس يسهل لك العمل على العديد من الدوال والكلاسات complex subsystems فكل من الكلاسات الأخرى قد يكون بها الكثير من الدوال ولكن عن طريق وضعهم داخل ال Facade class فقد يمكن أن يقلل ال API لمستخدم هذه الكلاسات وبالتالي يقلل التعقيدات في الاستخدام. للمزيد انظر لصفحة الويكيبيديا

مثلاً لديك جدول في القاعدة البيانات عن المستخدمين وبه كل معلوماتهم واسم المستخدم وكلمة المرور والخ، في حال أرجعت ذلك ال Model إلى المستخدم في صفحة البحث فهذا يعني أنك ارجعت كل شيء بما فيها معلومات حساسة لمستخدم آخر، لذلك يمكنك أن تقوم بعمل UserViewModel وترجع فيه معلومات المستخدم التي تكون متاحة للجميع فقط مثلا الاسم والصورة وتستطيع ايضاً وضع متغيرات تحصل عليها

بعد حسابات ما، مثلاً تضع عدد الردود الكلية، عدد المواضيع، وتقوم بتعبئة هذه الخانات من جدول المواضيع والردود، وبالتالي هذا الViewModel سوف يعرض وفيه كافة المعلومات.

سوف نقوم فيما يلي بعمل ProductViewModel في هذه الطبقة:

```
1.  namespace Products.Service
2.  {
3.      public class ProductViewModel
4.      {
5.          public int ProductId { get; set; }
6.          public string Name { get; set; }
7.          public string RRP { get; set; }
8.          public string SellingPrice { get; set; }
9.          public string Discount { get; set; }
10.         public string Savings { get; set; }
11.     }
12. }
```

الكلاينت لهذه الطبقة (وهو ال Presentation Layer) عندما يتعامل مع ال Service Layer سوف يرسل لها نوع العميل ويرجع النتيجة، وبدلاً من ارسال ال CustomerType الذي قمنا بعمله سابقاً فسوف نستخدم Pattern اسمه Request-Response Messaging Pattern وهذا يجعلنا نتعامل مع مفهوم ال Request بعض النظر عن المعاملات المرسلة فيه ونوعها وبالتالي لن تتغير ال API بتغير هذا الكلاس.

كلاس ال Request وفيه سوف يكون نوع العميل

```
1.  namespace Products.Service
2.  {
3.      public class ProductListRequest
4.      {
5.          public CustomerType CustomerType { get; set; }
6.      }
7.  }
```

كلاس النتيجة ProductListRespone وسوف ترجع قائمة المنتجات بالإضافة إلى متغير يمثل نجاح أو فشل العملية مع رسالة يكون بها نص يفيد في حال الفشل.

```
1.  namespace Products.Service
2.  {
3.      public class ProductListResponse
4.      {
5.          public bool Success { get; set; }
6.          public string Message { get; set; }
7.          public IList<ProductViewModel> Products { get; set; }
8.      }
9.  }
```

مرة أخرى اتباع هذا ال Pattern سوف يسهل فيما بعد لو احتجت أن ترسل أمور إضافية مثلاً تريد تطبيق ال Paging وتريد عرض 20 منتج كل صفحة، فسوف ترسل رقم الصفحة التي تريدها وتضيفها مع ال Request والنتيجة Response قد ترجع فيها متغيرات أخرى مثلاً العدد الكلي، والمتبقي وهكذا.

الآن لدينا ال ProductViewModel في هذه الطبقة، والطبقة التي قبلها بها Product، فنريد طريقة لتحويل ال Model.Product إلى ProductViewModel والسبب كما ذكرنا أننا لا نريد أن نرجع ال Model.Product

إلى ال Presentation وسيتم استخدام ViewModel بحيث عندما تتغير الواجهه فيما بعد وتطلب معلومات اضافية (مثلاً تقييم المنتج) فالتغيير سوف يكون على ال ViewModel ولن نجري أي تغيير على ال Model.Product.

لإجراء التحويل يمكن أن نقوم به بأنفسنا، أو نستخدم مكتبات مساعدة للتحويل (تسمى Mapper Library) وحالياً سوف نقوم بالعملية بأنفسنا لأنها بسيطة والمشروع صغير، لكن في حال كثرت ال Entities وكل منها فيها الكثير من المعاملات فيمكنك أن تستخدم أي من المكتبات مثل Auto Mapper.

حالياً سوف تحتاج دالتين، دالة تحول كائن واحد ودالة تحول List من الكائنات وتقوم بعمل حلقة وتستخدم دالة تحويل الكائن.

يمكن أن نضيف هذه الكلاسات في ال Product Domain Model ولكن هي ليست حقاً Business Logic لذلك من الافضل وضعها ك Extension Method حتى تكون هذه الدوال كأنها ضمن ال Product. وسوف نقوم بإضافة الكلاس ProductMapperExtensionMethod كما يلي:

```
1.  namespace Products.Service
2.  {
3.      public static class ProductMapperExtensionMethods
4.      {
5.          public static IList<ProductViewModel> ConvertToProductListViewModel(this
6.                           IList<Product> products)
7.          {
8.              IList<ProductViewModel> productViewModels = new List<ProductViewModel>();
9.              foreach (Product p in products)
10.             {
11.                 productViewModels.Add(p.ConvertToProductViewModel());
12.             }
13.
14.             return productViewModels;
15.         }
16.
17.         public static ProductViewModel ConvertToProductViewModel(this Product product)
18.         {
19.             ProductViewModel productViewModel = new ProductViewModel();
20.             productViewModel.ProductId = product.Id;
21.             productViewModel.Name = product.Name;
22.             productViewModel.RRP = String.Format("{0:C}", product.Price.RRP);
23.             productViewModel.SellingPrice =
24.                       String.Format("{0:C}", product.Price.SellingPrice);
25.
26.            if (product.Price.Discount > 0)
27.             productViewModel.Discount = String.Format("{0:C}", product.Price.Discount);
28.
29.             if (product.Price.Savings < 1 && product.Price.Savings > 0)
30.                productViewModel.Savings = product.Price.Savings.ToString("#%");
31.
32.             return productViewModel;
33.         }
34.     }
35. }
```

الدالة الاولى مسؤولة عن تحويل مجموعه من ال Products إلى مجموعه أخرى من ال ProductViewModel وبالطبع يجب في الأخير أن تقوم بتحويل الحقول من كائن لآخر وهو ما يوجد في الدالة الثانية من الكلاس.

لاحظ عملية التحويل في الدالة الثانية، بعض الحقول تكون مباشرة بأخذ ما يقابلها مثلاً Name فهي توجد في الاثنين، بعض الحقول مثلاً Price لا يوجد على ال ViewModel ولكن تجزئت الحقول إلى Selling Price وDiscount وSaving.

وبهذا الشكل سوف تتم عملية التحويل من ال Model.Product إلى ال Service.ProductViewModel

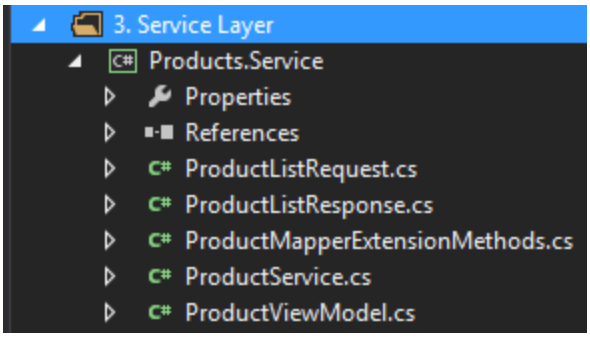
أخيراً نعرض القلب النابض لهذه الطبقة ألا وهي ال Service.ProductsSerivce والتي تتعامل مع ال Model.ProductsService وتجلب البيانات ومن ثم تقوم بتحويلها ال ProductViewModel

```
1.  namespace Products.Service
2.  {
3.      public class ProductService
4.      {
5.          private Model.ProductService _productService;
6.          public ProductService(Model.ProductService ProductService)
7.          {
8.              _productService = ProductService;
9.          }
10.
11.         public ProductListResponse GetAllProductsFor(ProductListRequest request)
12.         {
13.             ProductListResponse productListResponse = new ProductListResponse();
14.             try
15.             {
16.                 IList<Model.Product> productEntities = _productService
17.                         .GetAllProductsFor(request.CustomerType);
18.
19.                 productListResponse.Products =
20.                         productEntities.ConvertToProductListViewModel();
21.                 productListResponse.Success = true;
22.             }
23.             catch (Exception ex)
24.             {
25.                 productListResponse.Success = false;
26.                 productListResponse.Message = "An error occurred";
27.             }
28.             return productListResponse;
29.         }
30.     }
31. }
```

لاحظ في دالة البناء أن ال Service.ProductService يستقبل ال Model.ProductService ويعمل على اساسها، والكائن سوف يتم إرساله من الكود الذي يقوم بعمل كائن من ال Service.ProductService وبالتالي المسؤولية لن تكون على هذا الكلاس (بمعنى أن هذا ال Object تم إرساله Injected لهذا الكلاس).

دالة جلب المنتجات لا يوجد بها عمل كثير، سوى استدعاء دالة جلب المنتجات من طبقة ال Business من كلاس ال Model.ProductService ومن ثم تحويل المنتجات ال ProductViewModel ووضع الflag بأن العملية نجحت وإلا في حال الفشل فسوف يتم وضع رسالة الخطأ، وهذا مكان مناسب للتعامل مع الأخطاء، حتى يعلم الكلاينت بأن هناك مشكلة في ال Service Layer.

بهذا نكون أنهينا الطبقة الثانية وهذه هي شكلها في المشروع:



طبقة الوصول لقاعدة البيانات Data Access Layer

هذه الطبقة هي التي سوف تتعامل مع قاعدة البيانات أو ال Data Store عموماً، بالطبع هناك عدة طرق يمكن أن تتعامل بها مع قاعدة البيانات في الدوت نت منها باستخدام ADO.NET وفتح الاتصال المباشرة واستخدام ال SQL statements، ومنها ما يكون باستخدام ال ORMs مثلاً Entity Framework أو غيرها من المكتبات.

حالياً سوف نستخدم ال LINQ To SQL وهي بسيطة، فقط في مشروع ال Repository قم بالضغط بالزر الأيمن عليه واضف Add New Item ثم ستظهر نافذة سوف تختار Data من القائمة في اليسار ومن ثم على اليمين سوف تختار LINQ To SQL Data Context وسميه مثلاً Products.dbml.

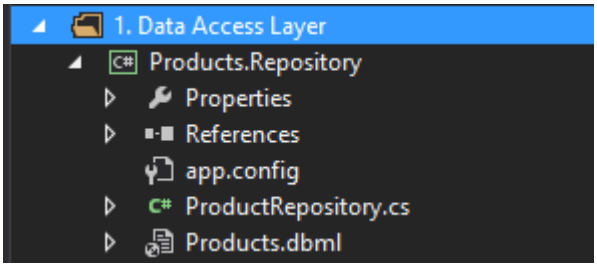
سوف يتولد بعدها الملف وستفتح نافذة ال Designer لها، حينها تستطيع عمل Connection مع القاعدة ومن ثم رمي الجدول Drag-And-Drop على شاشة ال Designer وحفظ الملف. وهكذا ستقوم بتوليد كلاس لكل جدول لديك في القاعدة يمثل ال Model لذلك الجدول. هذه الخطوات خارجة من هدف الموضوع وإنما هي فقط لمن يريد التطبيق ولكن الفكرة الآن أننا نريد التعامل مع قاعدة البيانات بطريقة ما.

الخطوة الوحيدة في هذه الطبقة هي عمل كلاس يمثل ال ProductsRepository ويطبق ال IProductsRepostiory Interface كما يلي:

```
1.  namespace Products.Repository
2.  {
3.      public class ProductRepository: IProductRepository
4.      {
5.          IList<Model.Product> IProductRepository.FindAll()
6.          {
7.              var products = from p in new ProductsDataContext().Products
8.              select new Model.Product
9.              {
10.                 Id = p.Id,
11.                 Name = p.Name,
12.                 Price = new Model.Price(p.RRP, p.SellingPrice)
13.             };
14.             return products.ToList();
15.         }
16.     }
17. }
```

ولاحظ أن الكائنات تم تحويلها إلى ال Model.Product (بالرغم من أن ال ORM تولد لك Model لكل جدول لكن لا يفضل أن تستخدمه ويفضل أن يبقى ذلك ال Model محصوراً داخل طبقة ال Repository).

شكل مشروع ال Repository الآن هو:



هكذا أنهينا 3 طبقات: ال Business Layer وال Service Layer وال Repository Layer ويمكن أن تستخدمها في التطبيق مباشرة ولكن نريد Layers اضافية بحيث نسمح لأكثر من كلاينت بالعمل، بالإضافة لتسهيل اختبار الشاشات، وسيتم اتباع ال Model View Presenter Pattern

طبقة العرض Presenter Layer

لتطبيق نمط ال MPV سوف نقوم بعمل interface يمثل كل الأشياء التي يمكن عملها على الواجهة وهي (طباعة المنتجات، سؤال المستخدم عن نوع العميل، طباعة الخطأ إن وجد) وهي في ال Interface التالي:

```
1.  namespace Products.Presentation
2.  {
3.      public interface IProductListView
4.      {
5.          void Display(IList<ProductViewModel> Products);
6.          Model.CustomerType CustomerType { get; }
7.          string ErrorMessage { set; }
8.      }
9.  }
```

هذا ال Interface سوف تطبقه الواجهة للبرنامج (سواء كان ال Console Application أو حتى الصفحة في الWeb Form Application). وتقوم بتعبئة الدوال بما يتناسب.

الكود الذي سيقوم بتعبئة الدوال هذه هو ال Presenter

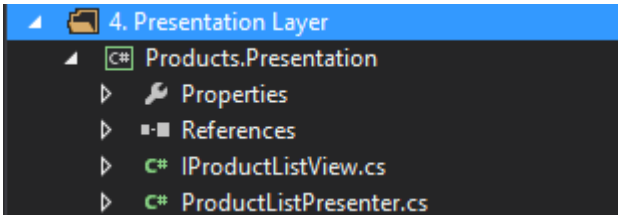
```
1.  namespace Products.Presentation
2.  {
3.      public class ProductListPresenter
4.      {
5.          private IProductListView _productListView;
6.          private Service.ProductService _productService;
7.
8.          public ProductListPresenter(IProductListView ProductListView,
9.                  Service.ProductService ProductService)
10.         {
11.             _productService = ProductService;
12.             _productListView = ProductListView;
13.         }
14.
15.         public void Display()
16.         {
17.             ProductListRequest productListRequest = new ProductListRequest();
18.             productListRequest.CustomerType = _productListView.CustomerType;
19.             ProductListResponse productResponse =
20.                 _productService.GetAllProductsFor(productListRequest);
```

```
21.
22.             if (productResponse.Success)
23.                 _productListView.Display(productResponse.Products);
24.             else
25.                 _productListView.ErrorMessage = productResponse.Message;
26.         }
27.     }
28. }
```

ال Presenter هو المسؤول عن جلب البيانات، التعامل مع احداث المستخدم، وتحديث ال View بالنتيجة. وهي تستخدم ال Service.ProductService بالإضافة ال Response-Request حتى تقوم بجلب المنتجات والخطأ أن وجد.

لاحظ أنه يتعامل مع ال IProductListView بغض النظر عن كونها صفحة ويب أو كلاس في كونسول، المهم انه سيقوم باستدعاء اشياء تطبقها الواجهة (مثلاً CustomerType في سطر 18 في الويب قد تكون جلب القيمة الموجودة على ال Dropdown List بينما في الكونسول سوف تكون سؤال المستخدم وجلب ما يكتبه)

طبقة ال Presentation تكون بالشكل التالي:



طبقة واجهة المستخدم User Experience Layer

وصلنا لآخر طبقة في المشروع وهي التي ستعرض الواجهة على المستخدم وتعرض البيانات. سوف نقوم بنوعين من الواجهات الأول ConsoleUI والثانية WebFormUI.

نبدأ بال ConsoleUI:

في مشروع ال Console Application سوف نقوم بعمل كائن من ال Presenter وكائن يمثل ال View وهو كلاس قمنا بعمله اسمه ConsoleUI. ومن ثم تم استدعاء دالة عرض المنتجات.

```
1.  class Program
2.  {
3.      private static ProductListPresenter _presenter;
4.
5.      static void Main(string[] args)
6.      {
7.          ConsoleUI console = new ConsoleUI();
8.
9.          _presenter = new ProductListPresenter(console,
10.             new Products.Service.ProductService(
11.               new Products.Model.ProductService(
12.                 new ProductRepository())));
13.
14.         _presenter.Display();
15.
16.         Console.ReadKey();
```

```
17.     }
18.
19.     class ConsoleUI : IProductListView
20.     {
21.         public void Display(IList<ProductViewModel> Products)
22.         {
23.             foreach (ProductViewModel product in Products)
24.             {
25.            Console.WriteLine(String.Format("Name: {0} \nPrice: {1:C} \nRRP: {2:C} " +
26.                                    "\nDiscount: {3} \nSaving: {4}\n",
27.                 product.Name, product.SellingPrice, product.RRP,
28.                         product.Discount, product.Savings));
29.             }
30.         }
31.
32.         public CustomerType CustomerType
33.         {
34.             get
35.             { return (CustomerType)Enum.ToObject(typeof(CustomerType),
36.                     int.Parse(Console.ReadLine()));
37.             }
38.         }
39.
40.         public string ErrorMessage
41.         {
42.             set
43.             {
44.                 Console.WriteLine(String.Format("Error: {0}\n", value));
45.             }
46.         }
47.     }
48. }
```

طريقة عمل الكائن اصبحت دسمة بعض الشيء، لاحظ كائن ال Presenter يتطلب كائن من كلاس يطبق ال IProductListView وهو الآن console، بينما المعامل الثاني يحتاج Service.ProductService. ولذلك قمنا بعمل كائن من ال Service.ProductService ولكن ذلك الكائن أيضاً يحتاج كائن من نوع Model.ProductService لذلك قمنا بعمله، والاخير يحتاج أي كائن يطبق ال IProductRepository لذلك أرسلنا كائننا الذي قمنا بعمله في طبقة ال Repository.

هذا الكود سليم ولكن الأصح هو استخدام ال Dependency Injection Container والتي تقوم بكل عمليات الإنشاء هذه بدلاً عنك وترسل الكائن وما يحتاجه من اعتماديات. لكن بما أننا لم نتحدث عن هذا الموضوع فالكود اعلاه يكفي.

تبقى الجزء الثاني وهو كلاس ال View الذي أسميناه ConsoleUI وتذكر أن هذه الدوال سيتم استدعائها بواسطة ال Presenter، وبالطبع دالة ال Display سوف تعرض البيانات على الشاشة، بينما دالة ال CustomerType (بالأصح ال Get Property) سوف يسأل المستخدم عن نوع العميل، بينما دالة طباعة رسالة الخطأ سوف تطبع الرسالة على الشاشة.

هذه الدوال أعلاه هي التي ستختلف من كلاينت لآخر، ولنرى كيف ستعمل على ال Web Form UI

تستطيع تشغيل المشروع هذه اللحظة (يمكنك تشغيل المشروع المرفق مع الكتاب) وستجد أن المخرج يعمل بنفس النتيجة التي ظهرت عندما كنا نستخدم الطريقة التقليدية.

مشروع ال Web Interface:

هذه مجرد لمحة بسيطة عن كيفية الاستفادة من الMVP وسترى أن العمل قليل في ال Web UI، ويمكنك تصفح هذه الفقرة بسرعه أو تخطيها إذا لم تكن مبرمج ASP.NET Web Form.

في ال Web Form UI فقط سنغير كود الثلاثة دوال السابقة حتى يعمل على الويب، مثلاً لطباعة رسالة الخطأ لن تكون على ال Console بل ستكون على Label موجود في الصفحة، بنفس الأمر سؤال المستخدم عن نوع العميل سوف يكون عن طريق معرفة القيمة المختارة من ال Dropdown List، وأخيراً عرض المنتجات لن تكون جمل طباعة وإنما يتم وضعها على جدول إما GridView أو الأفضل هو استخدام ال Repeater لذلك.

انشئ مشروع ال WebFormUI (واضف كل ال References لل Layers فيه) وفيه قم بعمل صفحة جديدة وليكن Defualt.aspx واضف الكود التالي في صفحة ال Designer داخل ال Form وهي صفحة تعرض كل البيانات داخل ال Repeater:

```
<form id="form1" runat="server">
<div>
    <asp:DropDownList AutoPostBack="true" ID="ddlCustomerType" runat="server">
            <asp:ListItem Value="0">Standard</asp:ListItem>
            <asp:ListItem Value="1">Trade</asp:ListItem>
    </asp:DropDownList>

    <asp:Label ID="lblErrorMessage" runat="server"></asp:Label>

    <asp:Repeater ID="rptProducts" runat="server">
        <HeaderTemplate>
            <table>
                <tr>
                    <td>Name</td> <td>RRP</td> <td>Selling Price</td>
                    <td>Discount</td><td>Savings</td>
                </tr>
                <tr>
                    <td colspan="5">
                        <hr />
                    </td>
                </tr>
        </HeaderTemplate>
        <ItemTemplate>
            <tr>
                    <td><%# Eval("Name") %></td>
                     <td><%# Eval("RRP")%></td><td><%# Eval("SellingPrice") %></td>
                    <td><%# Eval("Discount") %></td><td><%# Eval("Savings") %></td>
            </tr>
        </ItemTemplate>
        <FooterTemplate>
            </table>
        </FooterTemplate>
    </asp:Repeater>
</div>
</form>
```

الآن في الكود من الخلف Code Behind أضف:

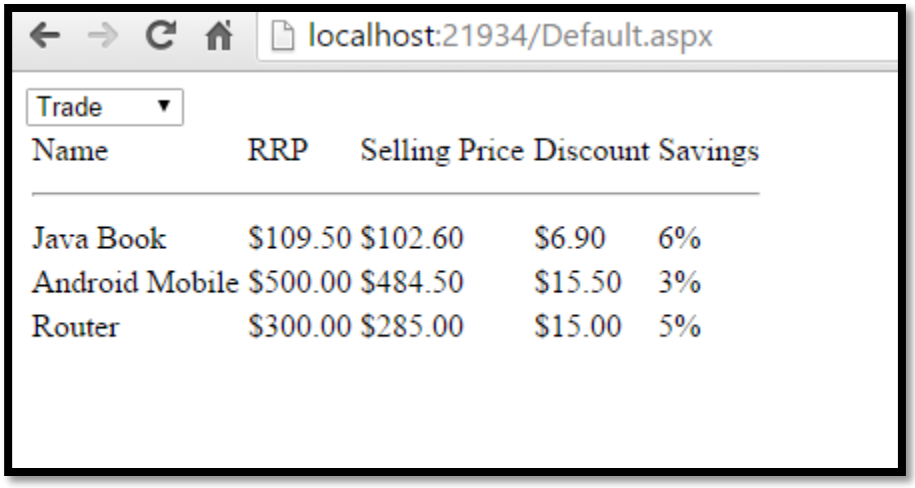
```
public partial class Default : System.Web.UI.Page, IProductListView
{
    private ProductListPresenter _presenter;

```

```
5.      protected void Page_Init(object sender, EventArgs e)
6.      {
7.          _presenter = new ProductListPresenter(this,
8.             new Products.Service.ProductService(
9.               new Products.Model.ProductService(
10.                 new ProductRepository())));
11.
12.         this.ddlCustomerType.SelectedIndexChanged +=
13.             delegate { _presenter.Display(); };
14.     }
15.
16.     protected void Page_Load(object sender, EventArgs e)
17.     {
18.         if (Page.IsPostBack != true)
19.             _presenter.Display();
20.     }
21.
22.     public void Display(IList<ProductViewModel> products)
23.     {
24.         rptProducts.DataSource = products;
25.         rptProducts.DataBind();
26.     }
27.
28.     public CustomerType CustomerType
29.     {
30.         get
31.         {
32.             return (CustomerType)Enum.ToObject(typeof(CustomerType),
33.                  int.Parse(this.ddlCustomerType.SelectedValue));
34.         }
35.     }
36.
37.     public string ErrorMessage
38.     {
39.         set {
40.             lblErrorMessage.Text =
41.             String.Format("<p><strong>Error</strong><br/>{0}<p/>", value);
42.         }
43.     }
44. }
```

لاحظ أن الكلاس نفسه قمنا بجعله يطبق ال IProductListView بمعنى هو ال View وعليه أن يعيد تعريف الدوال الثلاثة. دالة ال Page_Init سوف تعمل أول مرة وتقوم بعمل كائن ال Presenter بالطريقة نفسها التي بالكونسول وسبق وأشرنا أن ال DI Container هنا يفضل استخدامه. وفي دالة ال Load سوف يتم عرض المنتجات من خلال ال Presenter. وأخيراً الدوال 3 الباقية، وكل دالة تقوم بالمطلوب كما يتناسب مع طبيعة ال Web Page.

مخرج ال Web Form UI يكون كالتالي:

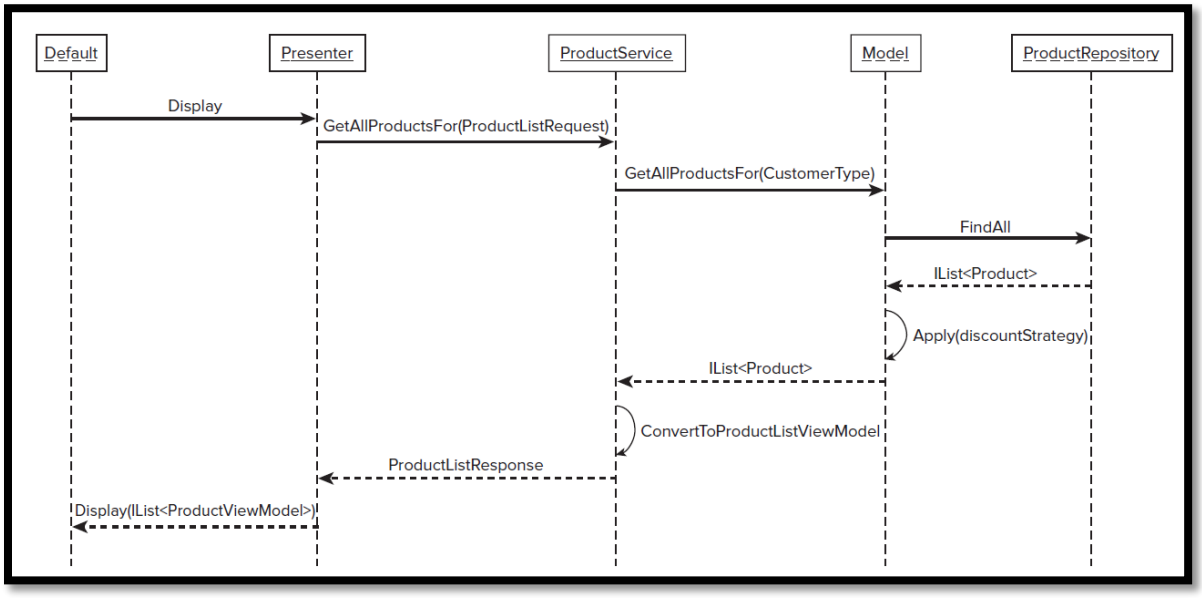


بالطبع تحسب النتيجة مباشرة بعد تغيير قيمة ال Dropdown لأنها هناك Listener عليها بمجرد أن يتغير يتم عمل Post Back للصفحة واظهار النتيجة الجديدة.

تهانينا!

قمنا بكثير من العمل فقط في هذا الفصل، ولكن النتيجة هو كود Loosley Coupled وقابل للاختبار وسهل الصيانة والعمل عليه في المستقبل، والفصل بين الطبقات فيه واضح.

المخطط التالي يوضح سير العمل بين مختلف الوحدات في هذا المشروع:



اختيار المعمارية المناسبة

هناك نقطة مهمة وسوف نوسع الحديث عنها في جزء ال Patterns، وهي في كون أنه لا توجد معمارية صحيحة أو خاطئة، وإنما توجد معمارية مناسبة ومعمارية أفضل منها، لذلك المهندس الجيد هو الذي يبنى المشروع بناء على المعطيات والمتطلبات التي لديه، فأغلب القرارات "الصحيحة" تبنى على حسب السياق،

فال context هو الذي يؤثر على القرارات التي نتخذها وليس اتباع النظريات وال best practices بشكل أعمى.

مثلاً مشروع صغير بميزانية صغيرة 100-1000 دولار في وقت ضيق (اسبوع إلى شهر) هل يجب أن استخدم Unit Testing مع ال Layered Architecture وأفصل كل شيء ب Interface وأقوم بعمل Implementation تحسباً للمستقبل؟

وفي الأساس نسيت أن المشروع ميزانيته 100 دولار والوقت المطلوب هو شهر؟ بهذا الشكل قمت بعمل ما يسمى over engineering وفي الغالب لن تستطيع اكمال المشروع في الوقت المحدد أو حتى بعدها بسبب تقعيده واحتماليه فشله صارت أكبر الآن بالرغم من أنك اعتمدت على ال best practices.

لكن في المقابل مشروع كبير سوف يمثل انطلاقة ل Business جديد مثلاً في فترة 2-6 أشهر ولديك فريق برمجي أو حتى بنفسك فكل شيء هنا يجب أن يبنى بطريقة مدروسة حتى تستفيد من فوائد الفصل وقابلية الاختبار وسهول العمل فيما بعد.

لذلك ال Real World Development/Architecture يجب أن تكون Pragmatic، تنظر لل Business كأول معامل، وتطور النظام بالاعتماد على ال Practical Consideration وليس فقط بمجرد تطبيق أي شيء تراه امامك ومع الممارسة والدراسة سوف يكتسب المهندس ذلك.

الفصل الخامس: خاتمة

استخدامات أخرى لل Interface

في هذا الكتاب تناولنا عدة استخدامات لل Interface ولنذكرها بسرعة:

- في مثال ال Regular Polygon الفصل الأول، استخدمنا ال Interface ك Abstraction وايضاً استخدمنا ال Abstract Class ورأينا أن ال Abstract Class أفضل منه والسبب أن هناك Shared Code مع لكل الابناء لذلك تم تفضيل طريقة ال Abstraction هذه. بقية الأمثلة التي تم طرحها فإن ال Interface كانت أفضل بكثير.
- مثال ال Repository Pattern سواءً قمنا بربط المكتبات بطريقة ال Compile Time Factory أو بتحميلها وقت التشغيل Dynamic Factory فال Interface أفضل حيث لا وجود لأي كود متشارك بين ال Implementations.
- مثال ال Discount في كلاس Price فلأن هناك أكثر من طريقة فتم استخدام ال IDicountStrategy وهناك أكثر من Implementation لها.
- في ال MVP فكل واجهة UI يجب أن تقوم بمجموعه من الامور وتم وضعها على ال IProductListView وكل ال UIs سوف تستخدم هذه ال Interface وتقدم Implementation لها، وهذه ال Interface سوف يستخدمها ال Presenter لكي يظهر المنتجات ويأخذ مدخل المستخدم ويظهر رسالة الخطأ أن وجدت.

ال Interface ودوره في العادات البرمجية الصحيحة

هل يقتصر استخدام ال Interface هنا؟ لا بالطبع أي خدمة يقدمها برنامجك ويحتمل أن يكون هناك أكثر من استخدام لها فهذا مثال جيد ل Abstraction هنا (سواءً كان Interface أو ال Abstract class). مثلاً لديك موقع وتريد أن تتأكد من اسم المستخدم وكلمة المرور Authentication، ولكن هناك أنواع من المستخدمين منهم من سوف تتأكد مباشرة من قاعدة البيانات ومنهم من سوف تتصل بال LDAP Server مثلاً Active Directory ومنهم من تكون بياناته في مكان آخر

فهنا يمكن أن تقوم بعمل IAuthentication ويكون لديك أكثر من ال Implementation لدالة ال Login، الأولى ال SQLAuthentication تستخدم الاتصال مع قاعدة البيانات، والثانية ال LDAPAuthentication تستخدم ال LDAP API لكي تتصل فيه، والأخيرة مثلاً ExternalAuthentication للنوع الثالث من المستخدمين.

والقرار المناسب (هل استخدم ال Interface أم ال Abstract Class) لتطبيق هذا ال Abstraction يكون على حسب الحالة والسياق.

ايضاً يعتبر ال Interface مفتاح أساسي في مواضيع أخرى مهمة مثل قابلية الاختبار Testability وسوف تعرف أنك تستطيع عمل Fake Implementation يطبق الInterface ويرجع بيانات تحددها سابقاً لكي تختبر دالة معينة بدلاً من جلب البيانات من مصدرها الحقيقي (لأنك تحتاج لتشغيل تلك الاختبارات مع كل Build للمشروع ويجب أن تكون سريعة في ثواني لذلك لن تتعامل مع مصدر البيانات الحقيقي)، وهذا ما يعرف بال Unit Testing وايضاً هناك ال Integration Testing والتي تهتم باختبار تلك المصادر الخارجية وتكاملها مع النظام ايضاً سواء كانت من قاعدة بيانات أو جلب بيانات من ويب سيرفس

ال Inversion of Control وال Dependency Injection أحد المواضيع التي تستخدم ال Interface بشكل أساسي، والتي تساهم في جعل الكود أكثر قابلية للاختبار وذلك لأنها تفصل الاعتمادية من الوحدات بشكل جيد Loosely Coupled.

أيضاً الكثير من العادات البرمجية Design Principles وال Design Patterns مثلاً ال Strategy تستخدم ال Abstraction وغالباً (ال Interface) كما تم طرحه في مثال الخصم في المنتجات.

حاولنا أن نكتب عن هذه المواضيع ولو بإيجاز بسيط ولكن وجدنا صعوبة في إضافتها في هذا الكتاب، حيث أنها تحتاج لكتيبات ومواضيع منفصلة ولذلك رأينا أن تكون في إصدارات اخرى وهذا أسهل للقارئ ايضاً، فتابعنا في مؤلفات وانفورماتيك (وصفحاتنا على شبكات التواصل الاجتماعي) لتحصل على الأجزاء المتبقية من السلسلة.

كما أن باب المشاركة مفتوح ويمكنكم التواصل عبر البريد أو صفحة انفورماتيك عبر شبكات التواصل الاجتماعي حتى يتم ادارة العمل وتنسيقه بطريقة أفضل.

شكراً لوصولك لهذه النقطة، أرجوا أن يكون هذا الجزء من السلسلة مفيداً وأن تكون قد استمعت خلال قراءتك لهذا الكتاب.

حتى لقاء آخر استودعكم الله الذي لا تضيع ودائعه.

وجدي عصام عبد الرحيم
الثلاثاء 2016/Jan/5

ملحق 1: انشاء واستخدام المكتبات Dependencies

معظم المبرمجين يعرفوا فكرة المكتبات Libraries وأنك تستطيع استخدام مكتبة تقوم بعمل ما لكي توفر على نفسك عناء كتابتها مجدداً، وربما قد لا تستطيع كتابتها من الاساس (مثلاً مكتبات التشفير والتي تتطلب خبرة في الموضوع اولاً قبل الخبرة البرمجية). لذلك المكتبات هي الحل.

سوف نستخدم كلمة "Dependencies" كمرادف لكلمة مكتبات، والسبب احياناً قد تريد استخدام مكتبة مثلاً X ولكن هذه المكتبة نفسها تستخدم مكتبة اخرى Y، لذلك حتى تستخدم المكتبة صحيحاً يجب أن يتوفر لديك المكتبتين معاً X و Y. لذلك سوف تجد الأداة الجديدة Nuget اسمها Dependency Manager لأنها تقوم بتحميل المكتبات والمكتبات التي تستخدمها (كل الاعتماديات المطلوبة) بضغطة زر واحدة بدلاً من أن نقوم بها بنفسك. لذلك إذا صادفت كلمة اعتمادية Dependency فهي نفس مفهوم المكتبة التي تعرفها.

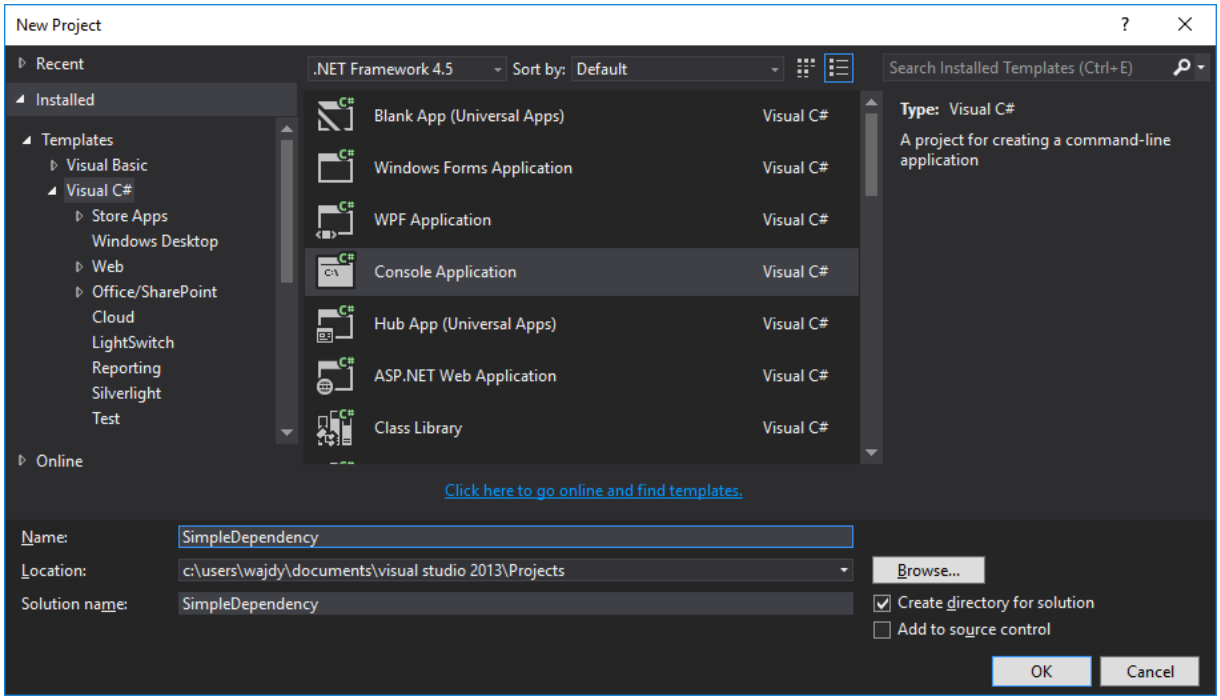
مقدمة لل Dependencies

كل البرمجيات بها Dependencies سواء كانت مكتبة داخل مشروعك First-Party Dependency أو مكتبة خارجية Third-Party Dependency أو حتى من مكتبة من ال .NET Framework. وفي الحقيقة أي مشروع يقوم بعمل شيء مفيد ففي الغالب يستخدم كل الأنواع الثلاثة

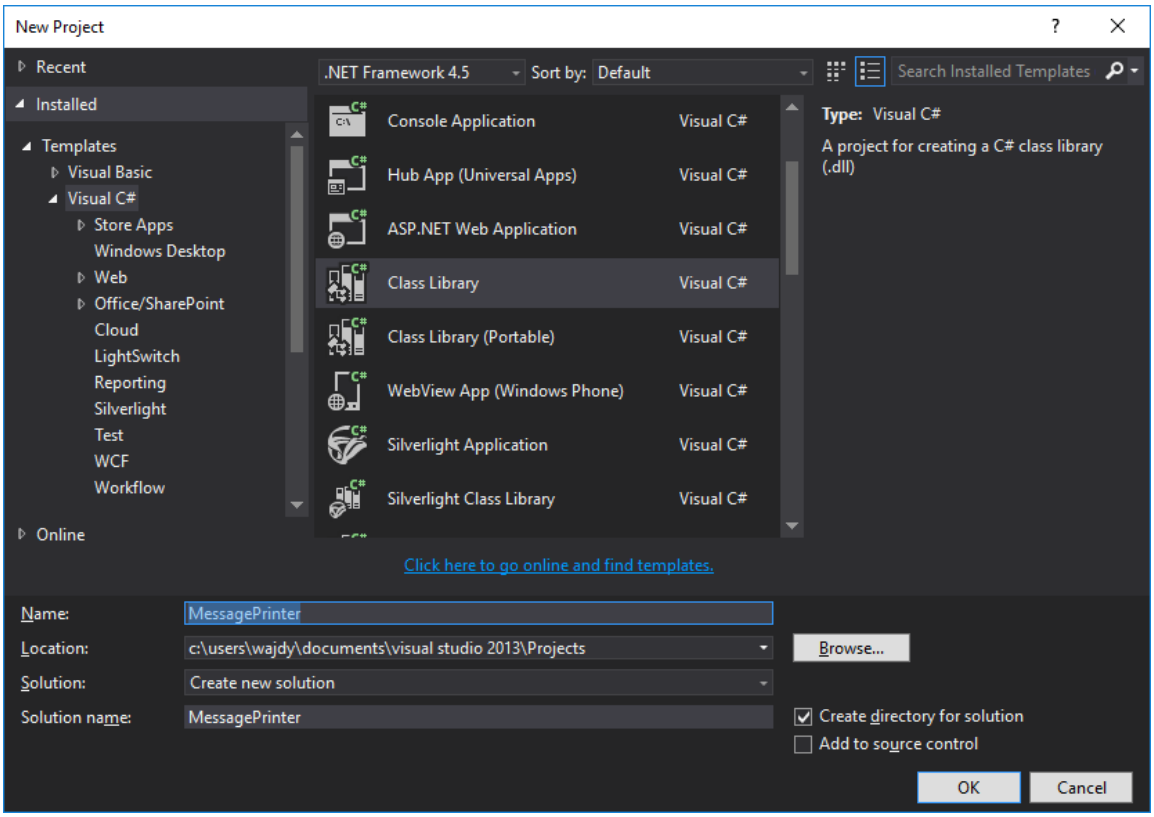
الاعتمادية تعني أن الكلاينت سوف يعتمد على B لإنجاز المهمة بدون الحاجة لمعرفة التفاصيل في كيفية أداء عملها، ومن المهم أن تعرف أنك إذا كنت تعتمد على B فهذا لا يعني أن B يعتمد عليك

سوف نأخذ مثال Hello World خطوة بخطوة لنبين مفهوم المكتبات والاعتمادية في ابسط شكل:

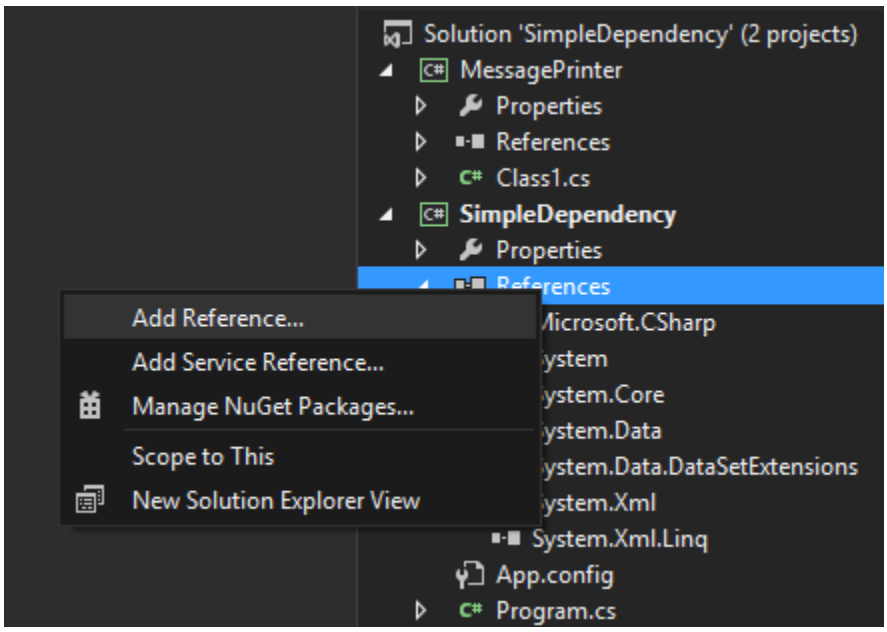
1) قم بفتح ال Visual Studio أي نسخة (المثال يستخدم 2013) وقم بعمل مشروع جديد واختر نوعه Console Application وليكن باسم SimpleDependency



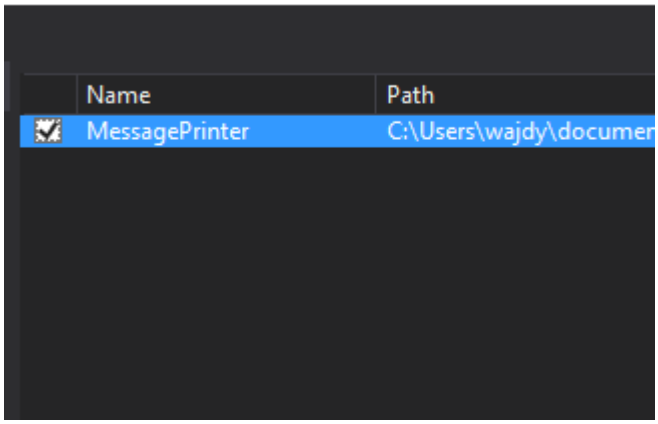
2) اضغط على ال Solution بالزر الأيمن واختر اضافةNew Project -> Add واختر نوعه ك Class Library وليكن اسمه MessagePrinter



3) اضغط بالزر الأيمن على ال Reference التي توجد في مشروع ال Console Application واختر اضافة Reference



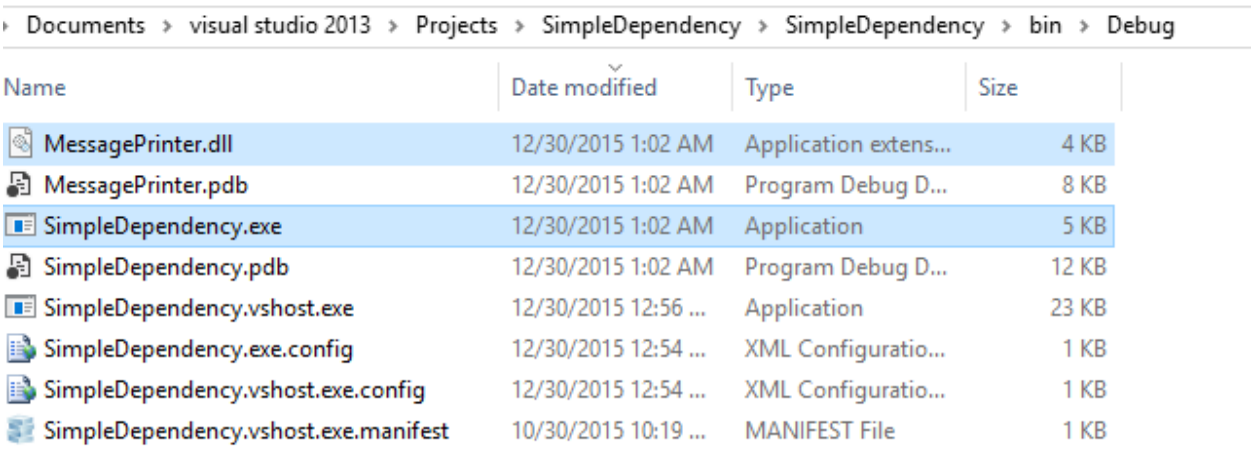
4) قم بإضافة ال MessagePrinter وسوف تشاهد الآن أنها أصبحت تحت ال References لل Console Application.



بهذا الشكل فقد أضفت ال Assembly (مكتبة) لل Console Project وأصبح هذا المشروع يعتمد على ال Class Library Project (ال MessagePrinter) ولكن ال MessagePrinter لا تعتمد على ال SimpleDenpendency.

5) قم ببناء المشروع Build وقم بفتح مجلد المشروع (اختر مشروع ال SimpleDependency بالزر الأيمن وقم باختيار Open Folder in File Explorer ثم اذهب لمجلد ال Bin وافتح اما ال Debug أو ال Release وذلك على حسب طريقة ال Build، إذا لم تغير شيء في الوضع الافتراضي هي Debug) سوف تجد الآن أن ال MessagePrinter.DLL داخل ذلك المجلد، حيث **يتم نسخ كل ال DLLs لكل**

الاعتماديات التي يستخدمها المشروع ويتم وضعها في المجلد وذلك بواسطة ال Visual Studio اثناء عملية ال Build.

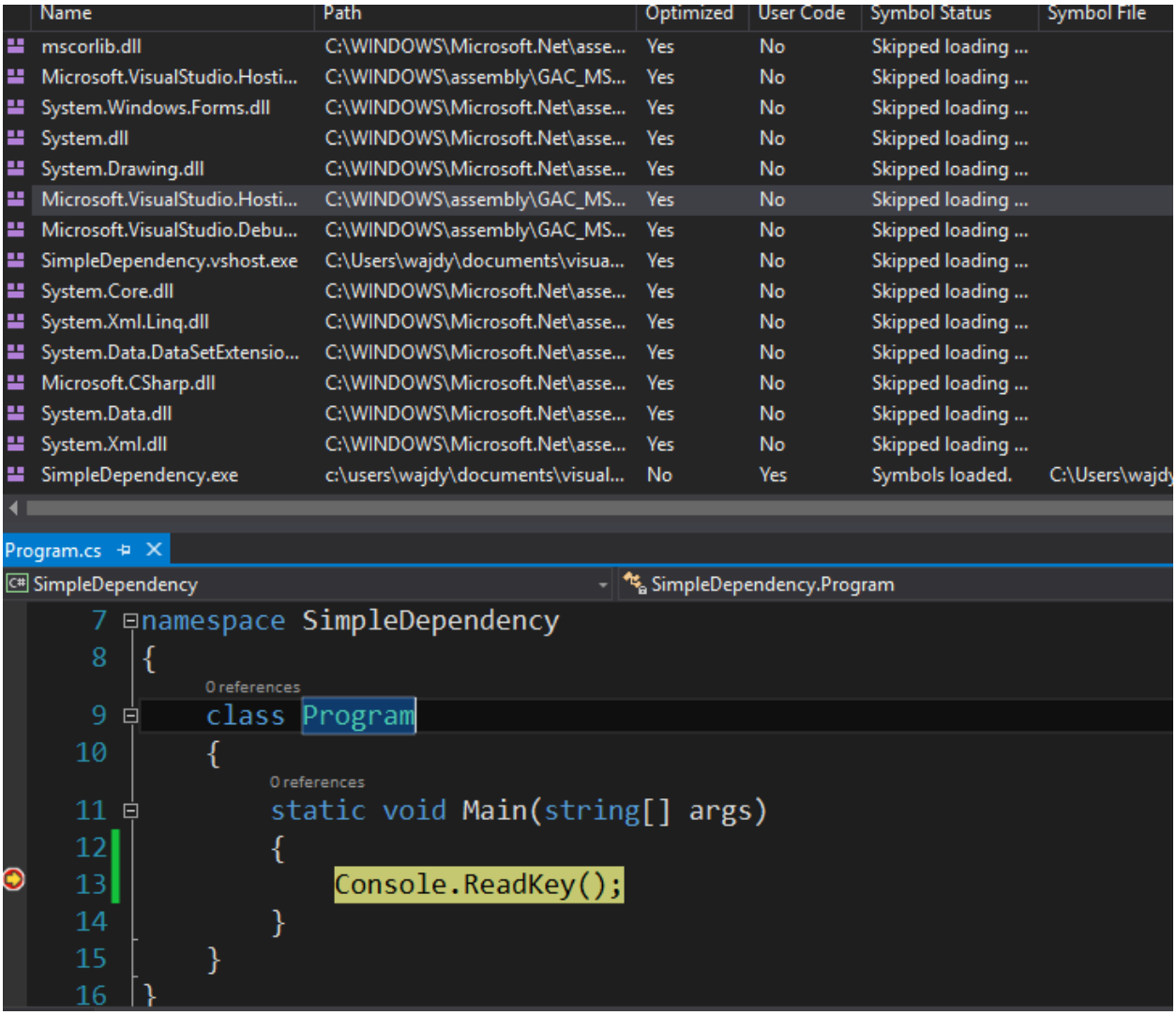


سوف نقوم الآن بعمل تعديل لأن المشروع ال SimpleDependency لا يعمل أي شيء وسوف يختفي مباشرة أن قمت بتشغيله ولذلك سوف نضع سطر قراءة حرف من المستخدم.

```
namespace SimpleDependency
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Console.ReadKey();
        }
    }
}
```

تمت الآن اضافة هذا السطر والذي سيمنع البرنامج من الانتهاء حتى تقوم بضغط أي زر، يمكنك تشغيله وادخال أي حرف وسينتهي البرنامج.

قم بوضع Breakpoint على هذا السطر Console.ReadKey وقم بتشغيل البرنامج وسوف يتوقف عند هذا السطر، نحن الآن نريد اظهار كل ال Assemblies التي تم تحميلها حتى هذه اللحظة، وتستطيع عرضها من خلال Debug -> Windows -> Modules وستخرج شاشة بها قائمة كل ال Assemblies التي تم تحميلها



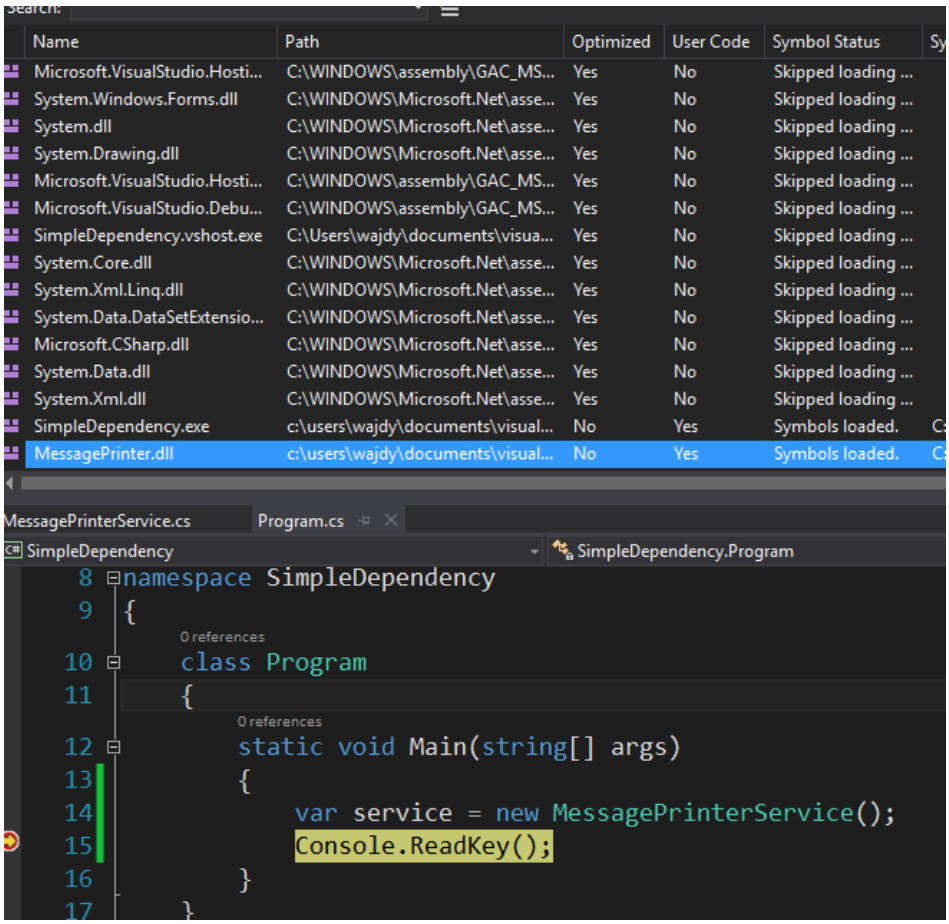
هل لاحظت شيء غريب؟ المكتبة التي اضفناها MessagePrinter لا توجد في هذه القائمة. والسبب هو أننا لم نستخدم أي شيء من داخل هذه ال Assembly حتى يقوم ال .NET Runtime بتحميلها!

قم بإيقاف التطبيق الأن، وعد لمجلد ال Bin وقم بحذف ال MessagePrinter.dll وشغل ملف ال SimpleDependency.exe في نفس المشروع وستجد انه يعمل ولا مشكلة فيه ولا وجود لأي Exception، **وهذا يثبت أن هذه ال DLL لم تستخدم خلال البرنامج**.

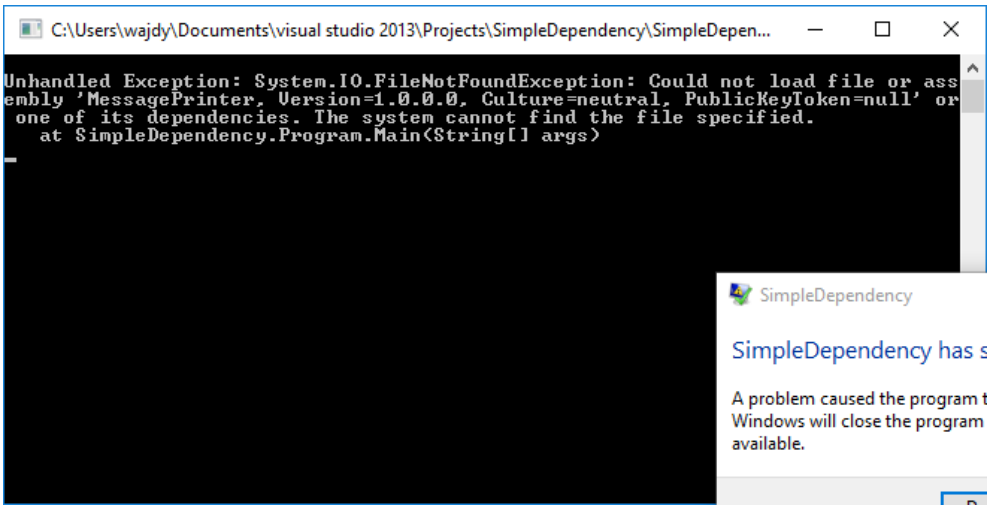
سنقوم ببعض الامور الاخرى حتى نفهم بالضبط ما الذي يجري، قم بفتح ال Program.cs واضف using MessagePrinter. هل تعتقد أن هذا السطر كفيل بجعل ال CLR يقوم بتحميل هذا ال Module؟ الجواب لا أيضاً.

سوف يتم تجاهلها ايضاً ولن يتم تحميلها (يمكنك التجربة والتأكد بنفس الخطوات السابقة) وذلك لأن ال using تعتبر Syntactic Sugar ووظيفتها هي بدل أن تقوم بكتابة ال Namespace كاملة لكل كلاس تريده فتقوم بعمل import لل Namespace واستخدام النوع مباشرة، وقبل عملية الترجمة أو المرحلة الاولى منها سوف يتم تبديل اسم النوع بال Full Namespace.

الآن اكتب جملة ال using وقم بعمل كائن من كلاس MessagePrinterService (قم بحذف class1 من مشروع المكتبة واضف هذا الكلاس) وهكذا عند تشغيل الكود والوقوف عند النقطة سوف تجد أن ال MessagePrinter.DLL قد تم تحميلها بطريقة صحيحة والسبب أننا استخدمناها



للتأكد ايضاً يمكنك أن توقف المشروع، وتقوم بمسح ال MessagePrinter.DLL الموجودة في المجلد Bin وعند تشغيل المشروع SimpleDependency.exe في نفس المجلد بعدها سوف تجد ال Exception التالي:



لل Class Library فوائد جمة في مشروعك، فقد يمكن أن تقسم المشروع لعدة طبقات Layers وكل طبقة تؤدي دور واحد مثلاً طبقة ال UI وهي مختصة بالعرض، طبقة ال Data Access وهي التي تختص بالتعامل مع قاعدة البيانات، وطبقة ال Business وهي التي بها الحسابات وسير العمل وكيفه تدفقه Workflows.

أحياناً قد تستخدم هذه الClass Library في أكثر من مشروع وبالتالي تضمن أن التغيير يكون في مكان واحد وهذا أفضل بكثير بدلاً من أن نقوم بالتغيير في عدة أماكن بسبب النسخ واللصق،

مثال: طلب منك عميل برمجة برنامج محاسبي على سبيل المثال، هذا البرنامج سوف يكون مقسم لجزئيين الجزء الأول الموقع الالكتروني وفيه كل الخدمات، بالإضافة إلى برنامج سطح مكتب Desktop لا يحتاج إلى اتصال بالإنترنت يقوم أيضاً بكل الخدمات.

عادة في المشاريع التي لها أكثر من Client (مثلاً موقع ولديه برنامج اندرويد او ايفون أو حتى Desktop لكن يجب أن يتصل بالإنترنت) فيفضل أن تبرمج خدمة ويب Web Service بحيث جميع الكلاينت وحتى الموقع يتعاملوا مع هذه الويب سيرفس وهي التي تأخذ وترجع النتيجة من والى الكلاينت.

لكن الطلب هذه المرة غريب قليلاً حيث كل الكود في الموقع (كل الخدمات) يجب أن تكون موجودة في ال Desktop Application والذي يجب أن يعمل بدون اتصال انترنت.

أحد الحلول هو نسخ كل الكود الذي يقوم بال Business (بالعمل والحسابات والخ) ولصقها على مشروع ال Desktop، وفقط سوف يكون الاختلاف هو في طريقة بناء ال UI، فهي تختلف بين الموقع وبرنامج سطح المكتب بكل تأكيد.

الحل الأفضل من السابق، هو وضع كل أكواد الحسابات والأمور المتعلقة بها في Class Library مثلاً اسمها Service Layer واستخدام هذه المكتبة في مشروع ال Web ومشروع ال Desktop وبالتالي الجميع يستخدم نفس الحسابات وكل شيء والتعديل إذا حدث –وسوف يحدث-سوف يكون في مكان واحد فقط.

ال Framework Dependencies

الاعتماديات أو المكتبات السابقة يمكن أن نطلق عليها First-Party Dependency فهي في نفس ال Solution مع ال Console Application وتستطيع الوصول لها وايضاً تغيير الكود إذا رغبت.

لكن أي مشروع تنشئه سواء كان Console Application أو Class Library Project أو حتى أي نوع آخر مثلاً Web Project سوف يكون له الاعتماديات المناسبة على حسب التطبيق (هناك ملف يحدد الAssemblies الافتراضية للمشروع وتستطيع تغيرها وحينها سوف يطبق على جميع المشاريع، ولكن في الغالب لا تحتاج لذلك) وهي تكون موجودة في ال NET Framework. الذي سيعمل على البرنامج (لذلك لا تنسخ هذه الاعتماديات إلى مجلد ال Bin).

لو لاحظت في المثال السابقة SimpleDependency سوف تجد أنه يعتمد على مجموعه من ال Assemblies اضيفت كلها عند عمل المشروع، على العموم للكود في المثال تستطيع حذفهم جميعهم ما عدا System وSystem.Core وسيعمل البرنامج فهو لا يستخدمهم (تم وضعهم لأنه هذا هو الشيء الافتراضي في أي مشروع).

ال NET Framework Assemblies. بعكس ال Dependencies الأخرى فهي دائماً يتم تحميلها بمجرد أن تقوم بعمل Reference لها حتى لو لم تستخدمها. لحسن الحظ إذا كان لديك أكثر من مشروع في نفس ال Solution

وكلهم يستخدموا نفس ال Assembly فسوف يتم تحميل نسخة واحدة فقط منها إلى الذاكرة ويتم استخدامها لهم جميعاً.

ال Third-Party Dependencies

هذا النوع يمثل المكتبات الخارجية التي يكتبها المطورين وتستخدمها، مثلاً بدلاً من بناء ORM لوحدك فتستطيع استخدام Entity Framework أو nHibernate وتوفر عليك جهد وعناء فترة طويل من العمل.
وبالطبع تستطيع اضافة ال Reference للمكتبات من هذا النوع من خلال تحديد مسار المكتبة من على جهازك، أو الحل الافضل من ذلك وهو عمل Folder داخل المشروع يمثل كل المكتبات ونسخ المكتبة عليه ومن ثم تحديد ال Reference من ذلك المجلد وبالتالي تكون كل المكتبات داخل المشروع وتكون داخل ال Source Control ايضاً.

والحل الأفضل من ذلك هو استخدام ال Nuget وهو Dependency Manager يسهل عملية اضافة ال Dependencies بالإضافة إلى تحديثها وعمل أي تغييرات مطلوبة في ال Configuration ايضاً.

في مواضع اخرى أن شاء الله سوف نتحدث عن ال Nuget وكيف يمكن أن تقوم بعمل مكتبة وتضيفها هناك بحيث تسهل على المبرمجين استخدامها.

ملحق:2 نظرة حول ال Explicit Interface في ال C#

عندما تقوم بعمل Implement لأي Interface وتقوم بالضغط على اسم الInterface واختيار CTRL+DOT سوف يظهر لك قائمة لكي تختار Implement Interface أو Explicit Implement Interface، في هذا الموضوع سوف نتحدث عن الفرق بينهم ومتى تحتاج لكي تقوم بعمل Explicit Implementation

الطريقة العادية في استخدام ال Interface هي في عمل كلاس يطبق تلك ال Interface وبعدها تكتب الكود، من ثم لكي تستخدم الكلاس تقوم بعمل كائن من نوع الكلاس وتضعه على نوع إما أن يكون من نفس الكلاس أو من ال Interface، كما يلي في الجزء الأيمن:

Declaration

```
public interface ISaveable
{
  string Save();
}
```

```
public class Catalog : ISaveable
{
  public string Save()
  {
    return "Catalog Save";
  }

  // Other members not shown
}
```

Usage

```
Catalog catalog = new Catalog();
catalog.Save(); // "Catalog Save"

ISaveable saveable = new Catalog();
saveable.Save(); // "Catalog Save"
```

لكن ما هو ال explicit implementation، ولنكمل بالكلاس السابق Catalog وندخل فيه دالة Save ثانية، ولاحظ أن القديمة ادخل معها ISaveable.Save وهو ما يعرف بال Explicit Implementation لأنها وضح بها انها الدالة الخاصة بذلك الInterface ، ولذلك السبب سوف تحصل على طريقة عمل مختلفة في الكلاس.

Declaration

```
public interface ISaveable
{
  string Save();
}
```

```
public class Catalog : ISaveable
{
  public string Save()
  {
    return "Catalog Save";
  }

  string ISaveable.Save()
  {
    return "ISaveable Save";
  }

  // Other members not shown
}
```

الآن بالإمكان عمل النوع بالطريقتين، لكن لاحظ الفرق الآن في الاستدعاء:

Concrete Type

```
Catalog catalog = new Catalog();
catalog.Save(); // "Catalog Save"
```

Interface Variable

```
ISaveable saveable = new Catalog();
saveable.Save(); // "ISaveable Save"
```

Cast to Interface

```
((ISaveable)catalog).Save();
// "ISaveable Save"
```

بمعنى إذا كان لديك النوع ISaveable فلا تستطيع استدعاء إلا ال Explicit Implementation بينما لو كان النوع هو من الكلاس فسوف تقوم باستدعاء الدالة التي في الكلاس.

لاحظ أيضاً عند إعادة تعريف الدالة ISaveable.Save يجب أن تكون بدون Access Modifier بمعنى بدون public والا سوف تحصل على خطأ بالترجمة

حتى لو كان الكلاس فقط فيه دالة واحدة Explicit Implementation فسوف تخفي هذه الدالة عندما يكون لديك متغير نوعه من نوع الكلاس، كما يلي:

Declaration

```
public interface ISaveable
{
  string Save();
}
```

```
public class Catalog : ISaveable
{
  string ISaveable.Save()
  {
    return "ISaveable Save";
  }

  // Save() deleted
  // Other members not shown
}
```

Concrete Type

```
Catalog catalog = new Catalog();
catalog.Save(); //**COMPILER ERROR**
```

Interface Variable

```
ISaveable saveable = new Catalog();
saveable.Save(); // "ISaveable Save"
```

Cast to Interface

```
((ISaveable)catalog).Save();
// "ISaveable Save"
```

لماذا تحتاج لعمل ال Explicit Implementation

في حال لديك أكثر من interface لهم نفس اسم الدالة ولهم نوع راجع return value مختلف فسوف نحتاج أن نطبق الاثنين، لذلك يمكن أن نطبق واحدة منهم بطريقة عادية، والأخرى بطريقة ال Explicit أو الاثنين معاً بطريقة ال Explicit. (في حالة كان هم نفس القيمة الراجعة ونفس التصريح فدالة واحدة تكفي لهم الاثنين).

Declaration

```
public interface ISaveable
{
  string Save();
}
```

```
public interface IVoidSaveable
{
  void Save();
}
```

Implementation

```
public class Catalog :
  ISaveable, IVoidSaveable
{
  string ISaveable.Save()
  {
    return "ISaveable Save";
  }

  void IVoidSaveable.Save()
  {
    // no return value
  }

  // Other members not shown
}
```

الوراثة في ال Inheritance

قد تستطيع بسهولة وضع IEnumerable<Person> في أي كود يحتاج ل IEnumerable والسبب لأن ال IEnumerable<T> يرث ال IEnumerable وبالتالي أي كلاس يطبق ال IEnumerable<T> عليه أن يعيد تعريف كل ما في ال IEnumerable

```
public interface IEnumerable<T> : IEnumerable
```

بمعنى أن ال Interface يستطيع استخدام ال Contracts في ال Interfaces الأخرى بدلاً من إعادة كتابتها مجدداً.

لذلك والأفضل دائماً أن تكون ال Interface متعلقة بمهمة في مجال معين وإذا احتجت أن تضيف اليها اشياء ليست متعلقة فيمكن أن ترثها في Interface أخر.

كيف تقوم بعمل تغييرات في ال Interface مثل إضافة أو حذف الدوال

إذا كنت تبنى مكتبة برمجية API أو أي Service يستخدمها مبرمجين، فالتغييرات التي سوف تجريها يجب أن تدرسها بعناية، لذلك لا يفضل أن تضيف شيئاً وذلك لأن حتى إضافة دالة سوف تجعل كل من يعتمد عليه بحاجة إلى إضافة تلك الدالة، وبنفس الشي حذف دالة قد يفسد على الكلاينت Breaking Code

لكن يمكن استخدام فكرة الوراثة من ال interface لكي تضيف دوال جديدة في ال interface بدون أن تفسد على الكلاينت، وفي نفس الوقت قد يساعدك وتكون أفضل وبالتالي الكلاينت يستخدم ال interface المناسب للمهمة فقط.

قائمة المصادر

- دورة C# Interfaces في موقع pluralsight
- كتاب Adaptive Code via C#: Agile coding with design patterns and SOLID principles
- كتاب Professional ASP.NET Design Patterns