**Decorator Pattern - نمط التحسين**

**تعريف**

يقدم الـ Decorator Pattern طريقة ديناميكية لإضافة متحولات State وتوابع جديدة Behavior إلى أحد الكائنات. ما يجعل هذا النمط مفيد في تطوير الأنظمة هو عدم معرفة الكائن أنه قد تم تحسينه بإضافة أشياء جديدة عليه. يكمن مفتاح تنفيذ هذا النمظ بجعل المحسناتDecorators ترث Inherit وتملك كائن Instance من الفئة Class الأصلية المراد تحسينها.

**شرح**

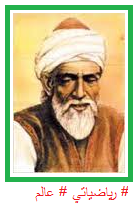
يوضح اسم النمط إمكانيته لإضافة تحسينات معينة على الكائن الأصلي, يمكننا أخذ كائن يقوم برسم صورة كمثال, حيث أنه بعد رسم الصورة يمكننا إضافة العديد من التحسينات عليها كرسم اطار حول هذه الصورة, أو إضافة وسوم خاصة بمحتويات تلك الصورة إضافة إلى كتابة عنوان لهذه الصورة.

[](http://1.bp.blogspot.com/-ipbdBzo3-xg/U9ACa3sDp7I/AAAAAAAAANI/Y-Ey2ayVAas/s1600/Untitled.png)

يوضح الشكل الصورة المرسومة بالكائن الأصلي من دون أي اضافات, باستخدام شيفرة مصدرية تشبه التالي

## IPhoto algo = new Photo();

## Algo.Draw();

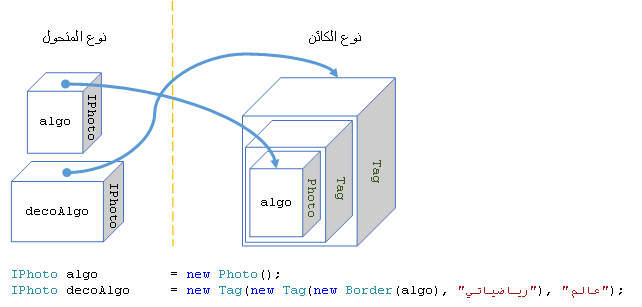
[](http://2.bp.blogspot.com/-8bOT-IBQHqU/U9ACtcg3GNI/AAAAAAAAANQ/r7d87jmO9ms/s1600/Untitled.png)

ويوضح الشكل الصورة المرسومة بعد استخدام مجموعة الكائنات المشكلة من الكائن الأصلي وكائنات التحسين (رسم الإطار وكتابة الوسوم), باستخدام الشيفرة المصدرية التالية وبالإستعانة بالشيفرة المصدرية السابقة:

IPhoto borderedAlgo = new Border(algo);  
IPhoto taggedPhoto = new Tag(new Tag(borderedAlgo, "عالم"), "رياضياتي");  
taggedPhoto.Draw();

يمكننا إنشاء عدد غير منتهي من الكائنات المحسنة وأستخدامها في تحسن الكائن الأصلي أو في تحسن كائن قد تم تحسينه مسبقاً, حيث يمكن استخدام هذه الكائنات بطريقة الدمج والتركيب لإنشاء نسخ محسنة من الكائن الأصلي.

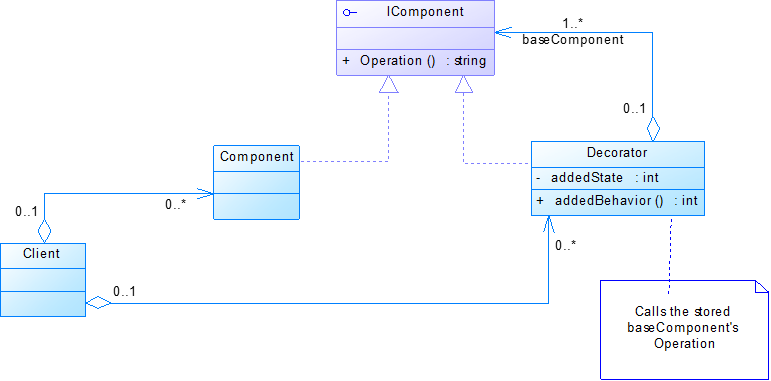
يوضح الشكل الثالث نوع المتحول إضافة إلى توليفة الكائنات التي تمثل نوع الكائن المراد إضافة تحسينات عليه.

[](http://2.bp.blogspot.com/-MdRvg6U3mM4/U9AC446XkRI/AAAAAAAAANY/rQHGSnjv1cs/s1600/Untitled.png)

**تأتي أهمية هذا النمط من أنه:**

* الكائن الأصلي ليس على دراية باي من الكائنات المحسنة
* لايوجد فئة واحدة تحوي جميع الميزات المراد إضافتها أو إزالتها من أو على الكائن الأصلي
* كافة التحسينات مستقلة عن بعضها البعض
* يمكن دمج وتوليف الكائنات المحسنة في بينها

**التصميم**

[](http://4.bp.blogspot.com/-OcyoW48v4k8/U9ADmEodAgI/AAAAAAAAANk/D4PFHIPXuaQ/s1600/Untitled.png)

يبين مخطط فئات الـ UML تصميم هذا النمط إضافة إلى العناصر التي تلعب الأدوار الرئيسية فيه وهي على الشكل التالي:

Component: الفئة الاساسية للكائنات التي يمكن إضافة تحسينات عليها أو إزالة تحسينات منها, (قد يوجد أكثر من واحدة من هذه الفئة تحقق نفس الواجهة).

Operation: تابع في كل الكائنات التي تحقق الواجهة IComponent يمكن أن يحتوي هذا التابع على تحقيقات مختلفة تبعاً للفئة. (قد يوجد العديد من هذه التوابع)

:IComponenet الواجهة التي تعرف فئات الكائنات التي يمكن تحسينها,Component أحدهم

Decorator: فئة تحقق الواجهة IComponent وتقوم بإضافة متحولات وتوابع جديدة (قد يوجد أكثر من واحدة من هذه الفئة)

يتضمن مخطط فئات الـ UML نوعين من العلاقات مع الواجهة IComponent:

1.      التحقيق وتدل على أن فئة الـ Decorator تحقق Implement الواجهة IComponent مايعني في حقيقة الأمر أن الكائنات من الفئة Decorator يمكن استخدامها في أي مكان يتوقع كائنات من النوع IComponent.  
  
الفئة Component أيضاً على علاقة تحقيق مع الواجهة IComponent ولذلك بإمكان الفئة Client استخدام الكائنات من النوع Component والكائنات من النوع Decorator بشكل تبادلي, وهنا يكمن جوهر هذا النمط.

2.      التجميع Aggregate: وتدل على أن الفئة Decorator تملك كائن أو أكثر من النوع IComponent, وتدل أيضاً على أن الكائنات المحسنة سوف تستخدم الكائنات الأصلية عند التنفيذ, تكمن الطريقة التي يستخدمها هذا النمط من أجل تنفيذ مهامه في أن الفئة Decorator تستخدم الخاصية baseComponent لكي تستدعي التابع Operation والتي ترغب في اعادة تعريفهOverride.

لاحظ أيضاً أننا أضفنا addedBehavior كتابع وaddedState كخاصية على الفئة Decorator كي نوضح أنه يمكن توسيع هذه الفئة بما يلزم من الأعضاء Members الأضافية.

**فحص**

صل العناصر الأساسية في الـ Decorator Pattern مع مثال تحسين الصور الموضح في الشرح.

قم بتغطية العمود الأيمن في الجدول وحاول تعريف العناصر كما تم توضيحها فقرة الشرح, ثم قم بالتأكد من صحة اجاباتك في العمود الأيسر.

|  |  |
| --- | --- |
| أي صور  صورة بسيطة  لعرض الصورة  صورة ذات وسوم أو اطار  كائن إنشاء الصور و الصورة ذات الوسوم أو الطارات | IComponent Component Operation Decorator Client |

من الجدول السابق يمكننا أن نرى أن سطور التعليمات التالية صحيحة داخل الفئة Client إذا أرادت وضع وسمين على صورة بسيطة:

Photo photo = new Photo();

Tag mathTag = new Tag(photo, "رياضياتي");

Tag scTag = new Tag(mathTag, "عالم");

سنرى في أغلب الأنماط أن العناصر التي تلعب أدوار رئيسية يمكن أن تأخذ عدة أشكال ولكي نحافظ على مخططات ال UML بسيطة لن نوضح كل هذه الأشكال عليها, لكن يجب أن نأخذ تأثيرات هذه الأشكال على التصميم بعين الأعتبار:

**تعدد المكونات**

تعرف الواجهة IComponent كافة الفئات التي تقبل التحسين - وهنا تكمن أهمية هذه الواجهة حتى لو لم تكن تحوي على أي تابع - وبذلك يمكن لكافة المحسنات أن تركب على كافة الفئات التي تحقق هذه الفئة, حيث بالإمكان أن توجد فئات أخرى تقوم برسم الأشخاص أو المنازل أو المركبات وكل هذه الفئات يمكن أن تضاف عليها أطار مثلاً أو وسم, أما في حالة التأكد أنه لن يكون هناك أي فئة أخرى مثل الفئةComponent في المستقبل عندها فقط يمكننا الإستغناء عن الواجهة IComponent وجعل الفئة Decoratoer ترث مباشرة من الفئةCommponent.

**تعدد المحسنات**

كما أنه بإمكاننا إنشاء عدة كائنات من النوع Tag لكي تقوم بتحسين الكائن من النوع Photo فإنه بإمكاننا إنشاء عدت فئات اخرى تقوم بتحسين الكائنات من النوع Photo أو من النوع Tag على حدى سواء, فمثالاً يمكننا إنشاء فئة محسنة تقوم برسم اطار على كائن من النوعPhoto أو على أي من كائناته المحسنة, أو إنشاء فئة محسنة تقوم بجعل الصورة ضبابية وإلى ما هنالك من هذه المحسنات. بغض النظر عن طبيعة الفئة المحسنة فإنها تملك كائن من النوع IComponent الذي بدوره يمكن أن يكون محسناً.

**تعدد التوابع**

تركز الشرح في السابق على الرسم كتابع أساسي والتوابع في الفئات الأخرى التي تقوم بتحسينه, لكن بالإمكان أن يكون هناك أكثر من تابع في الفئة الأساسية Component أو في الواجهة IComponent وأيضاً يمكن أن يتم تحسينها في الفئاتت Decorators كما نريد, إضافة إلى أن فئات التحسين يمكن أن تحتوي توسيعات كثيرة أخرى من التوابع أو الخصائص, ويمكن للفئة Client إستدعاء أي من هذه الأعضاء التي يمكنها الوصول إليها.

**التنقيذ:**

الميزة الأساسية في هذا النمط هي أنه لا يعتمد على الوراثة من أجل توسيع وتحسين إمكانياته, ولو توجب على الفئة Tag أن ترث الفئة Photoلتوسيع أو تحسين إمكانياتها لجعل ذلك من الفئة Tag فئة كبيرة وثقيلة ولربطها بالفئة Photo بقوة, لكن تحقيق الفئة Tag لنفس الواجهة التي تحققها الفئة Photo وبعد ذلك توسيعها بأعضاء جديدة أو بتحسين أعضائها الأساسية جعل منها فئة بسيطة سهلة القراءة والتعديل, إضافة إلى استطاعتها أن:

* تحقق أي عضو في الواجهة من جديد, أو تغير سلوك أي تابع من الكائن Component.
* إضافة أي خصائص أو توابع جديدة
* الوصول إلى أي من أعضاء الكائن الممر عند الإنشاء.

 يوضح المثال النظري التالي التفاعل بين فئات وكائنات هذا النمط وترتبط مباشر مع مخطط فئات الـ UML, يدخل في الأمثلة الحقيقة العديد من التحسينات والتوسيعات وحتى تغير في الأسماء ما يجعل من الصعب كشف هذا النمط.

namespace Decorator

{

    using System;

    public interface IComponent

    {

        string Operation();

    }

    public class Component : IComponent

    {

        #region IComponent Members

        public string Operation()

        {

            return "I am walking ";

        }

        #endregion

    }

    public class DecoratorA : IComponent

    {

        private IComponent baseComponent;

        public DecoratorA(IComponent c)

        {

            this.baseComponent = c;

        }

        #region IComponent Members

        public string Operation()

        {

            string str = baseComponent.Operation();

            return str + "and listening to Classical FM ";

        }

        #endregion

    }

    public class DecoratorB : IComponent

    {

        private IComponent baseComponent;

        public string addedState = "past the Coffee Shop ";

        public DecoratorB(IComponent c)

        {

            this.baseComponent = c;

        }

        #region IComponent Members

        public string Operation()

        {

            string str = baseComponent.Operation();

            return str + "to school ";

        }

        #endregion

        public string AddedBehavior()

        {

            return "and I bought a cappuccino ";

        }

    }

    class Program

    {

        static void Main(string[] args)

        {

            Console.WriteLine("Decorator Pattern\n");

            IComponent component = new Component();

            Display("1. Original Component: ", component);

            Display("2. Decoration-A : ", new DecoratorA(component));

            Display("3. Decoration-B : ", new DecoratorB(component));

            Display("4. Decoration-B-A: ", new DecoratorB(newDecoratorA(component)));

            // Explicit DecoratorB

            DecoratorB b = new DecoratorB(new Component());

            Display("5. A-B-decorated : ", new DecoratorA(b));

            // Invoke added state and added behavior

            Console.WriteLine("\t\t\t" + b.addedState + b.AddedBehavior());

        }

        private static void Display(string s, IComponent c)

        {

            Console.WriteLine(s);

            Console.WriteLine("\t" + c.Operation());

        }

    }

}

**الاستخدام**

سأسرد هنا أربع طرق لإستخدام الـ Decorator Pattern في أمثلة حقيقة

1.      كما شرحنا سابقاً فإن هذا النمط يناسب كثيراً عالم الصوريات, ويناسب أيضاً عالم الفيديو والصوت حيث يجب أن يتم خلط الفيديو مع الصوت القادم من خدمات الترجمة بشكل متزامن.

2.      كمثال حقيقي لهذا النمط في بيئة عمل .NET هو فئات الـ I/O حيث تمثل الفئات التالية الـ Component:

System.IO.Stream

   System.IO.BufferedStream

   System.IO.FielStream

   System.IO.MemoryStream

   System.Net.Sockets.NetworkStream

   System.Security.Cryptography.CryptoStream

والفئات التالي تمثل الـ Decorators:

System.IO.BinaryWriter

System.IO.StreamWriter

System.IO.BinaryReader

System.IO.StreamReader

لاحظ وجود الخاصية BaseStream داخل هذه الفئات حيث أنها تستخدم في عمليات القراءة الكتابة, داخل هذه الفئات.

3.      يمكن أيضاً أن نجد العديد من الأمثلة عن هذا النمط ضمن فئات الـ WPF حيث أن أغلب الفئات الموجودة في مجال الأسماءSystem.Windows.Controls يمكن أن يتم تحسينها وإضافة إمكانية جديدة لها من خلال الكائن Border  أو ViewBox

4.      يمكن أيضاً استخدام هذا النمط في رسم الشاشات في تطبيقات الويب حيث يمكن إنشاء كائن يقوم بتحسس نوع المستعرض وإذا كان جهاز محمول فإنه يرسم الشاشة بشكل صغير يتناسب مع حجم شاشة الجوال, وإذا كان جهاز كمبيوتر عادي يقوم برسم الشاشة بشكل كامل حيث أنه توجد مساحة كافية لذلك.

**نقوم باستخدام هذا النمط**

* عندما يكون لدينا فئات غير قابلة للوراثة
* عندما نريد إضافة خصائص وتوابع للكائن بطريقة ديناميكية
* عندما نريد التعديل على خصائص أو توابع كائن معين دون التأثير على الكائنات الأخرى من نفس النوع
* عندما نريد تجنب الوراثة لأن العديد من الفئات الموروثة سوف تنتج من ذلك

**ملاحظات:**

* يكمن الفرق الجوهري بين التحسين باستخدام الـ Decorator Pattern وبين الوراثة الاعتيادية هي أن المحسن يقوم بإضافة التحسينات على كائن معين بحد ذاته في حين أن الوراثة تقوم بإضافة التحسينات على الفئة كاملة, ومن هذا الفرق تأتي المرونة التي يتمتع بها الـ Decorator Pattern.
* تصبح عملية تصحيح الأخطاء أصعب عند استخدام هذا النمظ وذلك لأنه يقوم بإضافة التحسينات على الكائنات في زمن التشغيل