Patrones de Diseño

Arias, Cancino, Crispin, Gutierrez, Zuñiga

October 13, 2020

Abstract

Resumen

Los patrones de diseño son soluciones a problemas de diseño recurrentes en el desarrollo de software. Estos patrones se usan para mitigar la ausencia de algunos aspectos de calidad inherentes al software. Los desarrolladores, debido a su formación profesional, abordan diseños poco flexibles y difíciles de mantener en sus aplicaciones. Además, en ausencia de un lenguaje común con ingenieros de software se hace muy compleja la comunicación y validación del dominio de aplicación. Normalmente, las representaciones de los patrones de diseño se basan en diagramas de UML u otro tipo de grafos. Estos diagramas son difíciles de entender para los científicos e ingenieros de software inexpertos debido a su nivel de formalismo y, además, porque sólo representan el patrón de diseño aplicado y no el problema genérico que resuelven. Por otro lado, estos diagramas como unidad no poseen los elementos necesarios para representar completamente un dominio de software científico y se deben valer de la combinación de varios de ellos para hacerlo.El objetivo de esta investigación es describir lo que es un patrón de diseño, sus objetivos y clasificaciones aportando ejemplos de cada uno de ellos

Abstract

Design patterns are solutions for recurrent design problems in scientific software. These patterns are used to mitigate the lack of several quality aspects inherent in the software. Scientists, due to their professional training, tackle little flexible and maintainable designs for their software applications. In addition, in the absence of a common vocabulary with software engineers, domain communication and validation become complex. Normally, design patterns representation is based on UML class diagrams or other kind of graphs. These diagrams are difficult to understand for scientist and inexperienced software engineers due to their level of formalism and, also because of this diagram only represents the implemented design pattern and not the generic problem the design patterns solves. Furthermore, these diagrams as unity do not have the necessary elements to represent scientific software domains completely, so they must combine to do it. The objective of this research is to describe what a design pattern is, its objectives and classifications, providing examples of each of them

I. Introduccion

Como programadores seguramente nos habremos dado cuenta de que muchas veces resolvemos un mismo problema de manera diferente. Esto al principio de nuestra carrera puede ser algo positivo: experimentamos diferentes formas de enfocar un problema y podemos comparar los pros y los contras de nuestras anteriores soluciones, pero con el paso del tiempo nos gusta ir directamente al grano, y aplicar la solución más escalable, más testeable y más reutilizable. Por supuesto el no tener que explicar a cada compañero cómo hemos resuelto el caso es también algo deseable, pero esto no lo solucionan los patrones (a menos que nuestro compañero también los conozca) [5]

II. Desarrollo

i. Patrones de Diseño de Software

Son formas “estandarizadas” de resolver problemas comunes de diseño en el desarrollo de software[6].

Las ventajas del uso de patrones son evidentes:

- Conforman un amplio catálogo de problemas y soluciones
- Estandarizan la resolución de determinados problemas
- Condensan y simplifican el aprendizaje de las buenas prácticas
- Proporcionan un vocabulario común entre desarrolladores
- Evitan "reinventar la rueda"

Patrones de diseño creacionales

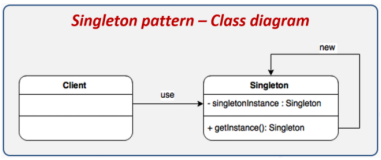
Como su nombre indica, estos patrones vienen a solucionar o facilitar las tareas de creación o instanciación de objetos[1].

Estos patrones hacen hincapié en la encapsulación de la lógica de la instanciación, ocultando los detalles concretos de cada objeto y permitiéndonos trabajar con abstracciones[1]. Los patrones creacionales son:

- **Abstract Factory:**Provee una interfaz o una clase abstracta, la cual define la creación de una familia de objetos relacionados o dependientes sin especificar su clase de origen.
- **Builder:** Se enfoca en la creación de objetos complejos. Este patrón separa el proceso de construcción del objeto de su implementación, logrando así usar el mismo proceso de creación para otras representaciones del mismo objeto.
- **Factory Method:** Define una interfaz genérica para la creación de un objeto. Este patrón de diseño trabaja a nivel de clases y le permite a una subclase instanciar la clase adecuada para el contexto definido.
- **Prototype:** Define una estructura deseada para el objeto y permite instanciar objetos haciendo una copia de éste.
- **Singleton:** Restringe el número de instancias de una clase a sólo una en todo el sistema.

PATRON SINGLETON

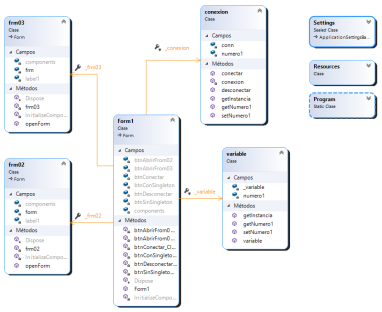
- **INTENCION:**
  Garantiza que una clase sólo tenga una instancia y proporciona un punto de acceso global a ella[7].
- **PROBLEMA:**
  Varios clientes distintos precisan referenciar a un mismo elemento y queremos asegurarnos de que no hay más de una instancia de ese elemento.
- **SOLUCION :**
  Garantizar una única instancia.



- **Participantes :**
  - **Client:**
    Componente que desea obtener una instancia de la clase Singleton.
  - **Singleton:**
    Clase que implementa el patrón Singleton, de la cual únicamente se podrá tener una instancia durante toda la vida de la aplicación[7].
- **Aplicabilidad :**
  Usar cuando:
  - Deba haber exactamente una instancia de una clase y ésta deba ser accesible a los clientes desde un punto de acceso conocido.
  - La única instancia debería ser extensible mediante herencia y los clientes deberían ser capaces de utilizar una instancia extendida sin modificar su código[8].
- **Consecuencias:**
  - Acceso controlado a la única instancia. Puede tener un control estricto sobre cómo y cuando acceden los clientes a la instancia.
  - Espacio de nombres reducido. El patrón Singleton es una mejora sobre las variables globales.

– Permite el refinamiento de operaciones y la representación. Se puede crear una subclase de Singleton.
– Permite un número variable de instancias. El patrón hace que sea fácil cambiar de opinión y permitir más de una instancia de la clase Singleton[8].

EJEMPLO DE IMPLEMENTACIÓN



CODIGO

```
2 referencias | chano, Hace 2 días | 1 autor, 1 cambio
public frm02 _frm02 { get; set; }
1 referencia | chano, Hace 2 días | 1 autor, 1 cambio
private void btnAbrirFrom02_Click(object sender, EventArgs e)
{
    _frm02 = frm02.openForm();
    _frm02.Show();
}
```

```
2 referencias | chano, Hace 2 días | 1 autor, 1 cambio
public frm03 _frm03 { get; set; }
1 referencia | chano, Hace 2 días | 1 autor, 2 cambios
private void btnAbrirFrom03_Click(object sender, EventArgs e)
{
    _frm03 = new frm03();
    _frm03.Show();
}
```

```
namespace Patron_Singleton
{
    7 referencias | chano, Hace 2 días | 1 autor, 1 cambio
    public partial class frm02 : Form
    {
        private static frm02 form = null;

        1 referencia | chano, Hace 2 días | 1 autor, 1 cambio
        private frm02()
        {
            InitializeComponent();
        }

        1 referencia | chano, Hace 2 días | 1 autor, 1 cambio
        public static frm02 openForm()
        {
            if(form == null)
            {
                form = new frm02();
            }
            return form;
        }

        1 referencia | 0 cambios | 0 autores, 0 cambios
        private void frm02_FormClosing(object sender, FormClosingEventArgs e)
        {
            form = null;
        }
    }
}
```

```
namespace Patron_Singleton
{
    4 referencias | chano, Hace 2 días | 1 autor, 2 cambios
    public partial class frm03 : Form
    {
        1 referencia | chano, Hace 2 días | 1 autor, 2 cambios
        public frm03()
        {
            InitializeComponent();
        }
    }
}
```

Patrones de diseño estructurales

Los patrones estructurales nos ayudan a definir la forma en la que los objetos se componen[1]. Los patrones estructurales son:

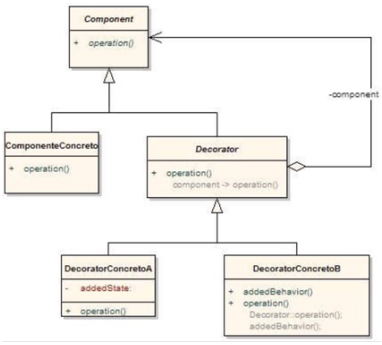
- **Adapter:** Permite a un cliente trabajar con clases cuya interfaz no posee una interfaz permitida en un sistema específico. Este patrón adapta la interfaz del objeto externo al de las clases del sistema al que se integra.
- **Bridge:** Desacopla la abstracción de una clase de su implementación, permitiendo así, que varíen independientemente y se entregue flexibilidad y facilidad de mantenimiento al software.
- **Composite:** Compone objetos relacionados en una jerarquía de tipo árbol. Este patrón de diseño le permite al cliente tratar de la misma manera un objeto y un grupo de objetos.
- **Decorator:** Añade responsabilidades adicionales a un objeto en tiempo de ejecución.
- **Flyweight:** Permite apoyarse en la copia de objetos complejos en un sistema que requiere la instanciación de un gran número de objetos con una estructura similar.
- **Facade:** Provee una interfaz que representa las interfaces de los subsistemas del software. Además, facilita el uso de un sistema complejo con muchas relaciones directas al cliente.
- **Proxy:** Restringe el acceso a un objeto para evitar su creación o participación precoz en el sistema. Usualmente, trata a objetos complejos o de gran tamaño.

PATRON DECORATOR

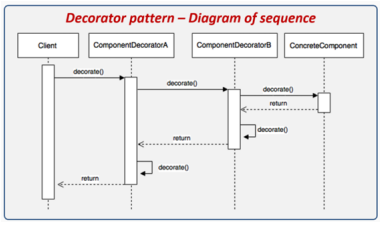
El patrón decorator está diseñado para solucionar problemas donde la jerarquía con subclasificación no puede ser aplicada, o se requiere de un gran impacto en todas las clases de la jerarquía con el fin de poder lograr el comportamiento esperado. Decorator permite al usuario añadir nuevas funcionalidades a un objeto existente sin alterar su estructura, mediante la adición de nuevas clases que envuelven

a la anterior dándole funcionamiento extra[1].

- **OBJETIVO :**
  Su función es determinar como las clases y objetos se combinan para formar estructuras. Estas estructuras permitirán que se agreguen nuevas funcionalidades.

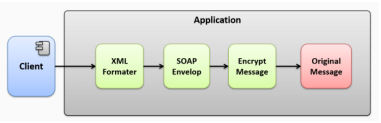


- **MOTIVACIÓN :**
  Dotar de funcionalidades dinámicamente a objetos mediante composición. Es decir, vamos a decorar los objetos para darles más funcionalidad de la que tienen en un principio. Esto es algo verdaderamente útil cuándo queremos evitar jerarquías de clases complejas. La herencia es una herramienta poderosa, pero puede hacer que nuestro diseño sea mucho menos extensible.
- **APLICACIONES :**
  El patrón se utiliza en los casos siguientes:
  - Añadir responsabilidades a objetos individuales de forma dinámica y transparente
  - Responsabilidades de un objeto pueden ser retiradas
  - Cuando la extensión mediante la herencia no es viable
  - Hay una necesidad de extender la funcionalidad de una clase, pero no hay razones para extenderlo a través de la herencia.
  - Existe la necesidad de extender dinámicamente la funcionalidad de un objeto y quizás quitar la funcionalidad extendida.

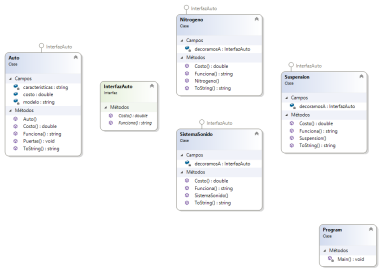


  - El Cliente realiza una operación sobre el DecoratorA.
  - El DecoratorA realiza la misma operación sobre DecoradorB.
  - El decoradorB realiza una acción sobre ConcreteComponente.
  - El DecoradorB ejecuta una operación de decoración.
  - El DecoradorA ejecuta una operación de decoración.
  - El Cliente recibe como resultado un objeto decorado por todos los Decoradores, los cuales encapsularon el Component en varias capas.

DIAGRAMA DE ARQUITECTURA :
Mediante la implementación del patrón de diseño Decorator crearemos una aplicación que nos permite procesar un mensaje en capas, donde cada capa se encargará de procesar un mensaje a diferente nivel.



- **EJEMPLO DE IMPLEMENTACION:**



CODIGO

```
public class Auto : InterfazAuto
{
    private string modelo;
    private string caracteristicas;
    public double costo;

    public Auto(string _modelo, string _caracteristica, double _costo)
    {
        modelo = _modelo;
        caracteristicas = _caracteristica;
        costo = _costo;
    }

    public void Puertas(bool _estado)
    {
        if (_estado) Console.WriteLine("Puertas cerradas");
        else Console.WriteLine("Puertas abiertas");
    }

    public override string ToString()
    {
        return string.Format("Modelo {0}, {1} \r\n", modelo, caracteristicas);
    }

    public double Costo()
    {
        return costo;
    }

    public string Funciona()
    {
        return "Encendiendo el motor";
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Negocios
{
    public interface InterfazAuto
    {
        double Costo();
        string Funciona();
    }
}
```

```
public class Nitrogeno : InterfazAuto
{
    private InterfazAuto decoramosA;

    public Nitrogeno(InterfazAuto _componente)
    {
        decoramosA = _componente;
    }

    public override string ToString()
    {
        return "Sistema de Nitrogeno\r\n" + decoramosA.ToString();
    }

    public double Costo()
    {
        return decoramosA.Costo() + 2500;
    }

    public string Funciona()
    {
        return decoramosA.Funciona() + ", Nitrógeno funcionando";
    }
}
```

Patrones de Diseño de Comportamiento

Los patrones comportamentales nos ayudan a definir la forma en la que los objetos interactúan entre ellos[1]. Los patrones de comportamiento son:

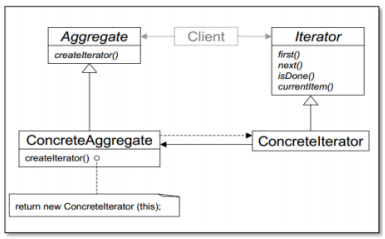
- **Chain of responsibility:** Evita el acoplamiento entre el objeto receptor y emisor de un mensaje específico. El patrón transfiere la responsabilidad a un objeto disponible.
- **Command:** Encapsula una petición como un objeto para permitir su parametrización.
- **Interpreter:** Permite definir una representación para su gramática y solucionar el mismo problema recurrente desde el contexto al que pertenece.
- **Iterator:** Proporciona una manera de acceder a un objeto agregado sin exponer su estructura interna.
- **Mediator:** Permite el bajo acople entre un conjunto de clases asignándole la implementación de los métodos a una sola clase.
- **Memento:** Captura y externaliza el estado interno de un objeto para que éste pueda variar y volver al estado inicial en cualquier tiempo.
- **State:** Permite a un objeto cambiar su comportamiento con base en los cambios de su estado interno.
- **Strategy:** Permite encapsular algoritmos o funcionalidades específicas y hacerlas intercambiables. Este patrón de diseño le permite al algoritmo variar sin depender del cliente.
- **Template Method:** Define el esqueleto de un algoritmo y permite cambiar ciertos pasos del mismo en las subclases. Estos cambios no afectan al algoritmo original.
- **Visitor:** Permite adicionar funcionalidades a una clase existente sin afectar su estructura.
- **Observer:** Define una dependencia de uno a muchos entre un conjunto de objetos para actualizarlos automáticamente una vez el objeto principal cambie su estado interno.

PATRON ITERATOR

Es un mecanismo de acceso a los elementos que constituyen una estructura de datos para la utilización de estos sin exponer su estructura interna. El patrón Iterator proporciona un acceso secuencial a una colección de objetos a los clientes sin que éstos tengan que preocuparse de la implementación de esta colección.

- **OBJETIVO :**
  Proporcionar una forma de acceder a los elementos de una colección de objetos de manera secuencial sin revelar su representación interna. Define una interfaz que

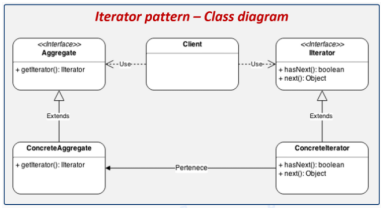
declara métodos que permitan acceder secuencialmente a la colección[3].



- **MOTIVACIÓN :**
  El patrón surge del deseo de acceder a los elementos de un contenedor de objetos (por ejemplo, una lista) sin exponer su representación interna. Además, es posible que se necesite más de una forma de recorrer la estructura siendo para ello necesario crear modificaciones en la clase. La solución que propone el patrón es añadir métodos que permitan recorrer la estructura sin referenciar explícitamente su representación. La responsabilidad del recorrido se traslada a un objeto iterador. El problema de introducir este objeto iterador reside en que los clientes necesitan conocer la estructura para crear el iterador apropiado
- **APLICACIONES :**
  El patrón se utiliza en los casos siguientes:
  - Es necesario realizar un recorrido de acceso al contenido de una colección sin acceder a la representación interna de esta colección.
  - Debe ser posible gestionar varios recorridos de forma simultánea.
- **CONSECUENCIAS :**
  Los iteradores simplifican la interfaz de las colecciones, ya que la interfaz de los recorridos se encuentra en los iteradores y no en la clase que corresponde a la estructura en cuestión. Permite variaciones en el recorrido de una colección. Para cambiar el algoritmo de recorrido basta cambiar la instancia de Iterator concreta Nuevos recorridos mediante nuevas subclases de Iterator Se puede tener más de un recorrido en progreso al mismo tiempo por cada colección

DIAGRAMA DE CLASES :
Los métodos más comunes son:

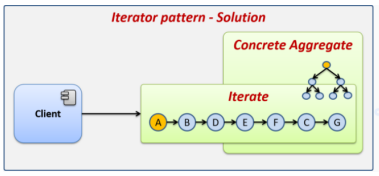
- **hasNext :** Método que regresa un booleano para indicar si existen más elementos en la estructura por recorrer. True si existen más y false si hemos llegado al final y no hay más elementos por recorrer.
- **next :** Regresa el siguiente elemento de la estructura de datos.



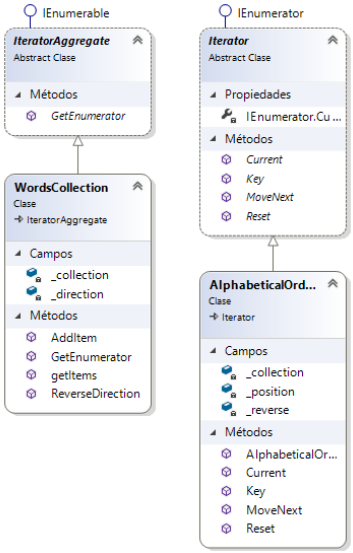
Los elementos del patrón Iterator se describen a continuación:

- **Client :** Actor que utiliza al Iterator.
- **Aggregate :** Interface que define la estructura de las clases que pueden ser iteradas.
- **ConcreteAggregate :** Clase que contiene la estructura de datos que deseamos iterar.
- **IIterator :** Interface que define la estructura de los iteradores, la cual define los métodos necesarios para poder realizar la iteración sobre el ConcreteAggregator.
- **ConcreteIterator :** Implementación de un iterador concreto, el cual hereda de IIterator para implementar de forma concreta cómo iterar un ConcreteAggregate.

- **DIAGRAMA DE ARQUITECTURA:**
  Patrón de diseño Iterator crearemos una aplicación que nos permita recorrer una estructura organizacional jerárquica, mediante la implementación de un iterador, el cual nos permitirá recorrer todo el árbol de la estructura de forma secuencial.



EJEMPLO DE IMPLEMENTACIÓN



CODIGO

```
abstract class Iterator : IEnumerator
{
    object IEnumerator.Current => Current();

    public abstract int Key();

    public abstract object Current();

    public abstract bool MoveNext();

    public abstract void Reset();
}

abstract class IteratorAggregate : IEnumerable
{

    public abstract IEnumerator GetEnumerator();
}
```

```
class AlphabeticalOrderIterator : Iterator
{
    private WordsCollection _collection;

    private int _position = -1;

    private bool _reverse = false;

    public AlphabeticalOrderIterator(WordsCollection collection, bool reverse = false)
    {
        this._collection = collection;
        this._reverse = reverse;

        if (reverse)
        {
            this._position = collection.getItems().Count;
        }
    }

    public override object Current()
    {
        return this._collection.getItems()[_position];
    }

    public override int Key()
    {
        return this._position;
    }

    public override bool MoveNext()
    {
        int updatedPosition = this._position + (this._reverse ? -1 : 1);

        if (updatedPosition >= 0 && updatedPosition < this._collection.getItems().Count)
        {
            this._position = updatedPosition;
            return true;
        }
        else
        {
            return false;
        }
    }

    public override void Reset()
    {
        this._position = this._reverse ? this._collection.getItems().Count - 1 : 0;
    }
}
```

```
class WordsCollection : IteratorAggregate
{
    List<string> _collection = new List<string>();

    bool _direction = false;

    public void ReverseDirection()
    {
        _direction = !_direction;
    }

    public List<string> getItems()
    {
        return _collection;
    }

    public void AddItem(string item)
    {
        this._collection.Add(item);
    }

    public override IEnumerator GetEnumerator()
    {
        return new AlphabeticalOrderIterator(this, _direction);
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {

        var collection = new WordsCollection();
        collection.AddItem("1");
        collection.AddItem("2");
        collection.AddItem("3");

        Console.WriteLine("Recorrido lineal:");

        foreach (var element in collection)
        {
            Console.WriteLine(element);
        }

        Console.WriteLine("\nRecorrido inverso:");

        collection.ReverseDirection();

        foreach (var element in collection)
        {
            Console.WriteLine(element);
        }

        Console.ReadKey();

    }
}
```

```
C:\Users\Wizzy\Desktop\UPT 20
Recorrido lineal:
1
2
3

Recorrido inverso:
3
2
1
```

III. Conclusiones

En este artículo se ha pretendido introducir al lector en el mundo de los patrones software, pero partiendo desde la perspectiva inicial de este concepto. Se ha buscado familiarizar al lector con el concepto de patrón, con sus características y sus peculiaridades, más que con su

utilización en el ámbito del Diseño Orientado a Objeto para construir aplicaciones software, con el fin de evitar caer en los falsos mitos y errores, causados por el desconocimiento y la falta de comprensión, propios de un acercamiento poco exhaustivo a una nueva área de conocimiento. Se ha querido recoger en este artículo una amplia lista de referencias, a las que cualquier lector interesado pueda acceder para profundizar más en este interesante campo.

IV. Recomendaciones

Al desarrollar aplicaciones robustas y fáciles de mantener, debemos cumplir ciertas "reglas". Lo pongo entre comillas porque aunque estas reglas de diseño son recomendables (muy recomendables), no son obligatorias. Siempre podemos decidir no aplicarlas. Aunque si no lo hacemos, hay que ser conscientes de la razón de no aplicarlas y de sus consecuencias. Los patrones de diseño nos ayudan a cumplir muchos de estos principios o reglas de diseño. Programación SOLID, control de cohesión y acoplamiento o reutilización de código son algunos de los beneficios que podemos conseguir al utilizar patrones.

V. Bibliografía

1 Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. EEUU: Addison-Wesley.

2 Coplien, James O. and Schmidt, Douglas (editors). "Pattern Languages of Program Design". Addison-Wesley. 1995.

3 Appleton, Brad. "Patterns and Software: Essential Concepts and Terminology". http://www.enteract.com/ bradapp/docs /patterns-intro.html. November 1997.

4 López Tallón, Alberto. "Patrones de Diseño. Reutilización de Ideas". Revista Profesional para Programadores (RPP). Nº43: 54-58. Septiembre, 1998.

5 Gamma, Erich, Helm, Richard, Johnson, Ralph and Vlissides, John. "Design Patterns. Elements of Reusable Object-Oriented Software". Addison-Wesley, 1995.

6 Alan Shalloway James R. Trott. (2004). Design Patterns Explained: A New Perspective on Object Oriented Design, 2nd Edition (Software Patterns). EEUU: Addison-Wesley.

7 Source Making. (2015). Singleton Design Pattern. Recuperado de https://sourcemaking.com/design_patterns/singleton

8 Tutorials Point. (2016). Design Pattern - Singleton Pattern. Recuperado de https://www.tutorialspoint.com/design_pattern/singleton_pattern.htm