UNIVERSIDAD CÉSAR VALLEJO

FACULTAD DE INGENIERÍA
Tecnología & Desarrollo
Vol. 12, Nº 1, Enero - Diciembre 2014, Pág.077-082

Aplicación de Patrones de Diseño para Garantizar Alta Flexibilidad en el Software

Applying Design Patterns to Ensure High Flexibility in the Software

Mg. Lain Cárdenas Escalante[1]

RESUMEN

Las aplicaciones software empresariales son por naturaleza complejas debido a su alto grado de variabilidad en sus reglas de negocio. Por tanto, el objetivo del artículo es explicar y demostrar en base a un ejemplo cómo el uso de patrones de diseño permite desarrollar aplicaciones empresariales flexibles a los cambios en sus reglas de negocio. En este trabajo se ha elaborado un modelo de diseño donde se aplica el patrón estrategia, el patrón compuesto y el patrón fábrica. Por último, también se recomienda el uso de algún framework que permita implementar la fabricación de objetos e inyección de dependencia.

Palabras Clave: Patrón de diseño, Modificabilidad, Estrategia, Bajo Acoplamiento, Alta Cohesión, Reglas de negocio.

ABSTRACT

Business software application is inherently complex because of its high degree of variability in its business rules. Therefore, the aim of this article is to explain and demonstrate, based on one example, how the use of design patterns can develop flexible business applications to changes in business rules. In this paper a Design Model based on a Strategy pattern, a Composite pattern and a factory pattern has been developed. Finally, the use of a framework that allows implementating the manufacture of objects and dependency injection is recommended.

Key Words: Design Pattern, Modifiability, Strategy, Low Coupling, High Cohesion, Business Rules.

Recepción: 29-05-2014 / **Aceptación:** 09-07-2014
[1]Universidad César Vallejo - Escuela de Ingeniería de Sistemas. Magister en Ciencias de la Computación. E-mail: laincardenas@gmail.com

1. INTRODUCCIÓN

Las empresas tienden por lo general a ir cambiando sus reglas de negocio, esto les ayuda a ser más dinámicos lo cual les permite competir con otras empresas y asegurar una mejor aceptación de sus clientes.

Por lo tanto, las aplicaciones de software empresarial deben ser capases de adaptarse a estos cambios del negocio, pero esta adaptación no debería ser traumático o complicado y generar mucha inversión de recursos y tiempo, es decir, el software debe ser flexible a los cambios.

En consecuencia, para lograr esta flexibilidad en el software, el software debe ser diseñado pensando en que va a sufrir cambios o modificaciones, sobre todo en sus reglas de negocio. Es aquí donde se debe hacer uso de patrones de diseño que nos ayudan a solucionar estos problemas.

El presente artículo explica y demuestra a través de un ejemplo el uso de algunos patrones de diseño como: estrategia, compuesto y fábrica que nos ayudan a desarrollar software garantizando su alta flexibilidad a los cambios.

Para el desarrollo y explicación de este trabajo, el artículo se ha organizado de la siguiente manera: en la Sección 2 se muestra un breve marco teórico; en la Sección 3 se describe el ejemplo con el uso de los patrones de diseño; y por último, en la Sección 4 se presentan algunas recomendaciones.

2. MARCO TEÓRICO

2.1 Extensibilidad

Este se centra en la extensión de un sistema de software con nuevas características, así como la sustitución de los componentes con las versiones mejoradas y la eliminación de las características y componentes no deseados o innecesarios. Para lograr extensibilidad un sistema de software requiere componentes débilmente acoplados. Se trata de una estructura que le permite intercambiar componentes sin afectar a sus clientes. El apoyo a la integración de nuevos componentes en una arquitectura existente también es necesario [1].

2.2 Modificabilidad

Estudio tras estudio muestra que la mayor parte del costo del sistema de software típico se produce después de que ha sido lanzado inicialmente. Los cambios se producen para agregar nuevas características, cambiar o incluso retirar los viejos.

Los cambios se producen para reparar defectos, reforzar la seguridad, o mejorar el rendimiento. Los cambios se producen para mejorar la experiencia del usuario. Los cambios se producen a adoptar nuevas tecnologías, nuevas plataformas, nuevos protocolos, nuevos estándares. Los cambios se producen para que los sistemas trabajen juntos, aunque nunca fueron diseñados para hacerlo.

Las tácticas para controlar la modificabilidad tienen como objetivo el control de la complejidad de hacer cambios, así como el tiempo y el costo de hacer cambios [2]. La Figura 1 muestra esta relación.

Figura 1. El objetivo de las tácticas de modificabilidad [2]

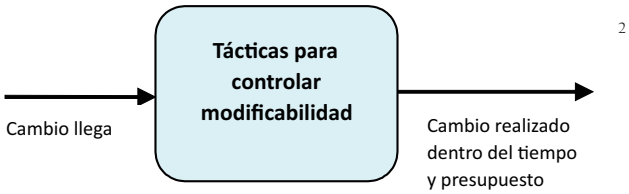
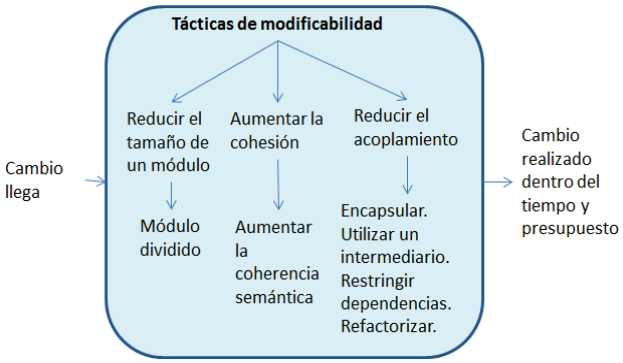


Figura 2. Tácticas de modificabilidad [2]



2.3 Patrones de diseño

Los patrones de diseño son descripciones de clases y objetos relacionados que están particularizados para resolver un problema de diseño general en un determinado contexto.
Un patrón de diseño nomina, abstrae e identifica los aspectos clave de una estructura de diseño común, lo que los hace útiles para crear un diseño orientado a objetos reutilizable. El patrón de diseño identifica las clases e instancias participantes, sus roles y colaboraciones, y la distribución de responsabilidades.

Cada patrón de diseño se centra en un problema concreto, describiendo cuándo aplicarlo y si tiene sentido hacerlo teniendo en cuenta otras restricciones de diseño, así como las consecuencias y las ventajas e inconvenientes de su uso [3].

2.4 Bajo acoplamiento

Hace parte de la clasificación de patrones GRASP.
Solución: Asignar una responsabilidad de manera que el acoplamiento permanezca bajo.
Problema: ¿Cómo soportar bajas dependencias, bajo impacto del cambio e incremento de la reutilización?

El acoplamiento es una medida de la fuerza con que un elemento está conectado a, tiene conocimiento de, confía en, otros elementos. Un elemento con bajo (o débil) acoplamiento no depende de demasiados otros elementos. Estos elementos pueden ser clases, subsistemas, sistemas, etcétera.

Una clase con alto (o fuerte) acoplamiento confía en muchas otras clases, por lo tanto son difíciles de reutilizar puesto que su uso requiere la presencia adicional de las clases de las que depende, además, que los cambios en las clases relacionadas fuerzan cambios locales.

El bajo acoplamiento soporta el diseño de clases que son más independientes, lo que reduce el impacto del cambio.

En general, las clases que son inherentemente muy genéricas por naturaleza, y con una probabilidad de reutilización alta, debería tener un acoplamiento especialmente bajo [4].

2.5 Alta cohesión

Hace parte de la clasificación de patrones GRASP.
Solución: Asignar una responsabilidad de manera que la cohesión permanezca alta.
En cuanto al diseño de objetos, la cohesión es una medida de la fuerza con la que se relacionan y del grado de focalización de las responsabilidades de un elemento. Un elemento con responsabilidades altamente relacionadas, y que no hace gran cantidad de trabajo, tiene alta cohesión. Estos elementos pueden ser clases, subsistemas, etcétera.

Una clase con baja cohesión hace muchas cosas no relacionadas, o hace demasiado trabajo. Tales clases no son convenientes; adolecen de los siguientes problemas: difíciles de entender, difíciles de reutilizar, difíciles de mantener, delicadas, constantemente afectadas por los cambios.

Una clase con alta cohesión es ventajosa porque es relativamente fácil de mantener, entender y reutilizar. El alto grado de funcionalidad relacionada, combinada con un número pequeño de operaciones, también simplifica el mantenimiento y las mejoras.

El grano fino de la funcionalidad altamente relacionada también aumenta el potencial e reutilización [4].

2.6 Factoría (Concreta)

Hace parte de la clasificación de patrones GoF.
Contexto/Problema: ¿Quién debe ser el responsable de la creación de los objetos cuando existen consideraciones especiales, como una lógica de creación compleja, el deseo de separar las responsabilidades de la creación para mejorar la cohesión, etcétera?
Solución: Crear un objeto Fabricación Pura denominado Factoría que maneje la creación [4].

2.7 Estrategia

Hace parte de la clasificación de patrones GoF.
Contexto/Problema: ¿Cómo diseñar diversos algoritmos o políticas que están relacionadas? ¿Cómo diseñar que estos algoritmos o políticas puedan cambiar?
Solución: Defina cada algoritmo/política/estrategia en una clase independiente, con una interfaz común [4].

2.8 Compuesto

Hace parte de la clasificación de patrones GoF.
Contexto/Problema: ¿Cómo tratar un grupo o una estructura compuesta del mismo modo (polimórficamente) que un objeto no compuesto (atómico)?
Solución: Defina las clases para los objetos compuestos y atómicos de manera que implementen el mismo interfaz [4].

3. APLICACIÓN DE LOS PATRONES DE DISEÑO

Para la aplicación de los patrones de diseño antes estudiados se tomará como ejemplo un caso relacionado a un sistema de pedidos de platos de comida de un restaurante.
Del caso de ejemplo, se ha considerado el Caso de Uso "Realizar Pedido". El propósito del Caso de Uso es registrar el pedido de un cliente en el cual se podrá seleccionar los platos que ofrece el restaurante e ingresar la cantidad que se desea de cada plato. El sistema debe permitir calcular el subtotal de cada línea de pedido y a la vez calcular el total que es la sumatoria de todos los subtotales de cada línea de pedido. Pero el negocio también ha considerado que se aplique un descuento al total del pedido siempre y cuando se cumpla alguna regla de negocio y por último calcular el monto a pagar, restando el total menos el descuento.

Analizando el requisito y las políticas del negocio, se sabe que las reglas para la aplicación del descuento pueden ir cambiando en el tiempo.

En consecuencia se suma un requisito no funcional que determina que el software debe soportar cambios en sus reglas y que esto no signifique un gran esfuerzo ni se vea afectado por el cambio.

Por ejemplo, se puede asumir que una primera regla/política de negocio para el cálculo del descuento está en función a un porcentaje del total, siempre y cuando el total haya superado un cierto monto.

Otra regla/política de negocio para el cálculo del descuento puede estar en función al total de platos solicitados, si se supera un cierto número de platos entonces se descuenta un monto determinado.

Otra regla/política de negocio para el cálculo del descuento puede estar en función a cada línea de pedido, si en una línea de pedido se ha solicitado una cantidad mayor a cierto número y corresponde a un plato con un precio superior a cierto valor, entonces se descuenta el valor de un plato. El descuento se sumaría por cada línea de pedido que aplique la regla.

También se podría aplicar otros tipos de descuentos que evalúen varias reglas de negocio y no sólo una.
Por ejemplo que calcule el descuento mayor o menor si el pedido aplica a más de una regla.

Por tanto, nos encontramos en un problema de diseño para dar solución a estos requerimientos. La solución de diseño que se dará aplicará el uso de los patrones descritos en el marco teórico.

3.1. Diseño de la solución

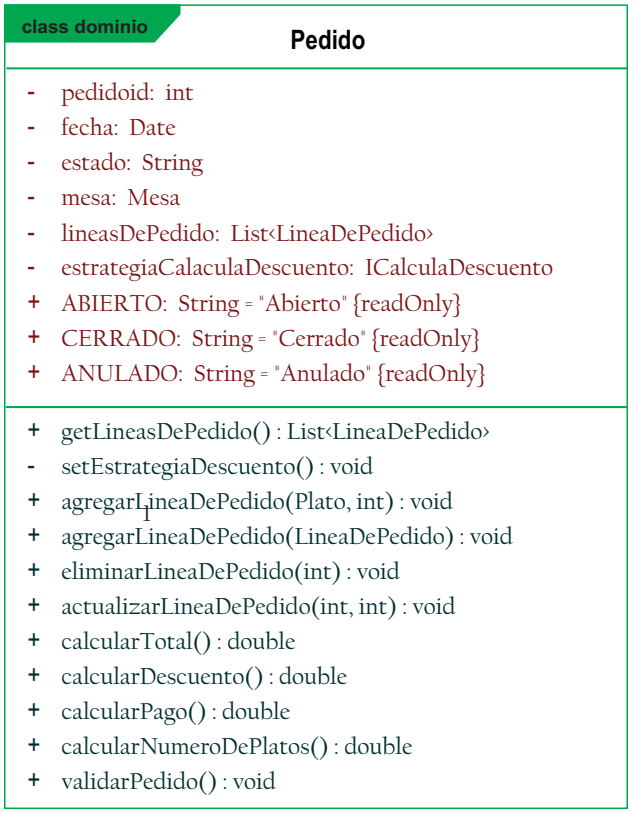
Para el diseño de la solución, en este artículo, sólo se ha considerado las clases que hacen parte de la capa del dominio dentro de la arquitectura, por ser la capa más importante en donde se resuelve la lógica de negocio.

La clase más importante del diseño para el caso de uso antes mencionado "Realizar pedido", es la clase pedido. Esta clase resuelve la mayor parte de las reglas de negocio asociados al pedido y hace cumplir el patrón alta cohesión, debido a que sus operaciones están estrechamente relacionadas.

La Figura 3 muestra la clase **pedido** con sus atributos y operaciones.

Como se observa, la clase pedido tiene una operación **calcularDescuento( )**, que es en la cual nos centraremos específicamente debido a que el Caso de Uso de "Realizar Pedido" debe poder resolver reglas asociadas al cálculo

Figura 3. Diseño de la clase pedido



del descuento y además como requisito no funcional asociado al caso de uso, se debe poder hacer cambios fácilmente en estas reglas.

En un diseño inicial se podría considerar la implementación del método **calcularDescuento( )**, que resuelva una regla de negocio. Pero el problema surge cuando esta regla de negocio deba cambiar.

En consecuencia se tendría que modificar la implementación del método para hacer cumplir la nueva regla. Esta forma de mantener o hacer los cambios en las reglas no es la más conveniente, provoca que se hagan nuevamente pruebas en la clase pedido, volver a compilar y desplegar la nueva versión del software. Se corre también el riesgo de agregar nuevos bugs.

Para solucionar este problema se aplica algunos principios y patrones de diseño. Por ejemplo, el Principio Abierto Cerrado, nos dice que las clases deben estar cerrados a los cambios y abierto a las extensiones. También nos basamos en el patrón polimorfismo, que permite implementar un método de diferentes formas. Estos están estrechamente relacionados, porque para hacer cumplir el principio abierto cerrado necesitamos del polimorfismo haciendo uso de clases abstractas o interfaces.
Por otra parte, se aplica el patrón estrategia para

implementar los diferentes algoritmos que representan las reglas de negocio para el cálculo del descuento. También se aplica el Patrón Compuesto en combinación con el Patrón Estrategia para dar solución a la implementación de las reglas compuestas, es decir de aquellas que evalúan varias reglas como por ejemplo encontrar el mayor o menor descuento. El uso de estos patrones de diseño también está estrechamente relacionado con el Patrón Polimorfismo y hacen cumplir el Principio Abierto Cerrado.

Por lo tanto, la clase Pedido en su operación calcularDescuento( ), ya no implementa una regla de negocio. En su lugar, la clase Pedido estará asociada a una interfaz que representa una abstracción de la estrategia que se desea implementar.

Esta interfaz declara la operación calcularDescuento(Pedido), que recibe como parámetro al objeto pedido. La interfaz es implementada por distintas clases donde cada clase implementa la operación definida en la interfaz, pero con un algoritmo distinto, es decir, cada clase resuelve una estrategia distinta para el cálculo del descuento. Entonces, la implementación de la operación calcularDescuento( ) de la clase Pedido invoca a la operación calcularDescuento(Pedido) de la clase estrategia que se haya asociado a la clase Pedido.

Por último, para poder asignar una clase estrategia a la clase Pedido, se hace uso del Patrón Factoría. El aplicar este patrón permite delegar en una clase fábrica la responsabilidad de crear la instancia de la clase estrategia que necesita la clase Pedido.

Con esto se evita una dependencia directa entre la clase Pedido y la clase estrategia que necesita. Así mismo, este tipo de solución está basado en el Patrón Bajo Acoplamiento.

Todo esto garantiza una fácil extensibilidad del software, ya que cada vez que se desee aplicar una nueva regla de negocio asociado al descuento, se creará una nueva clase que implemente la interfaz sin afectar ninguna parte del código ya implementado.

La Figura 4 muestra las clases estrategias simples que implementan la interfaz.

La Figura 5 muestra las clases estrategias compuestas que implementan la interfaz.

La Figura 6 muestra la clase **Pedido** y su relación con la interfaz y la clase **FabricaEstrategia**.

Figura 4. Clases estrategias simples

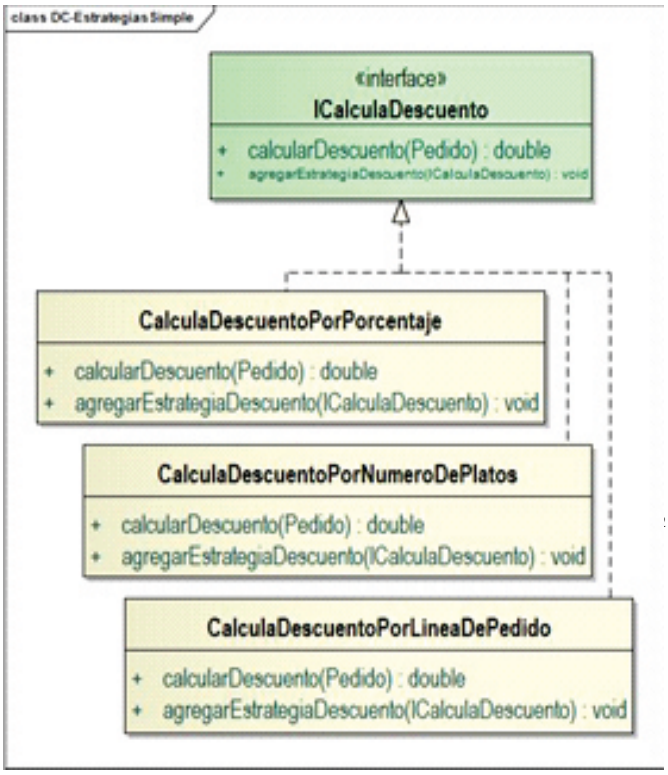


Figura 5. Clases estrategias compuestas

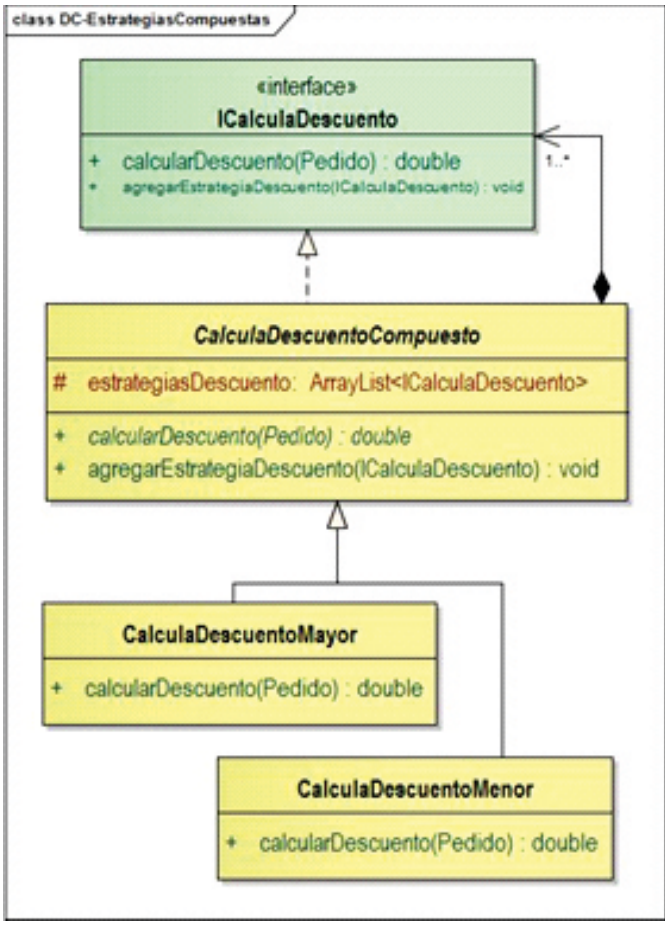
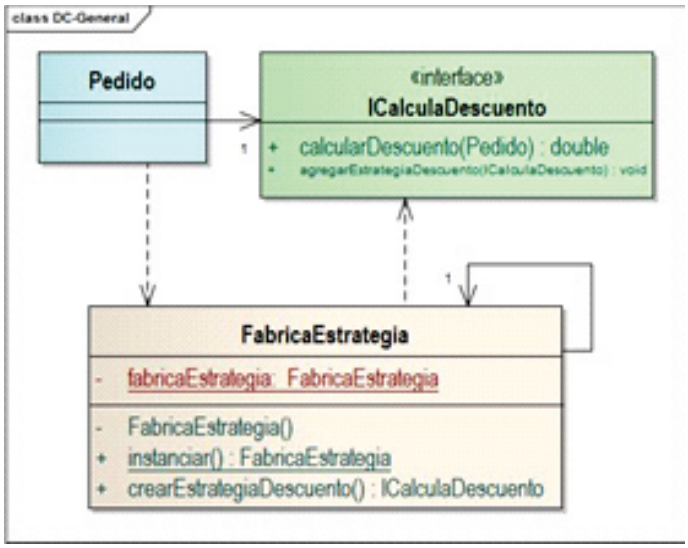


Figura 6. Clases relacionadas



3.2 Implementación de la solución

A continuación se mostrará algunas partes del código que demuestran la implementación de la operación calcularDescuento( ).

La Figura 7 muestra el código implementado del método **setEstrategiaDescuento ( )** de la clase **pedido**. Este método permite asignar la clase estrategia de descuento que la clase **pedido** va a usar, haciendo uso de la fábrica.

Figura 7. Método Set EstrategiaDescuento

```
private void setEstrategiaDescuento(){
      FabricaEstrategia fabricaEstrategia =
FabricaEstrategia.instanciar();
      estrategiaCalaculaDescuento =
fabricaEstrategia.crearEstrategiaDescuento();
}
```

La figura 8 muestra el código implementado del método **calcularDescuento( )** de la clase **pedido**.

Figura 8. Método calcularDescuento

```
public double calcularDescuento(){
      if(estrategiaCalaculaDescuento != null)
         return
estrategiaCalaculaDescuento.calcularDescuento(this);
      else
         return 0.0;
}
```

La figura 9 muestra el código implementado del método calcular Descuento(Pedio) de la clase CalculaDescuentoMayor.

Figura 9. Código implementado

```
@Override
public double calcularDescuento(Pedido pedido) {
      double descuentoMayor = 0.0, descuento;
      if(estrategiasDescuento.size()>0)
         descuentoMayor =
estrategiasDescuento.get(0).calcularDescuento(pedido);
      for(ICalculaDescuento estrategiaDescuento :
estrategiasDescuento){
         descuento =
estrategiaDescuento.calcularDescuento(pedido);
         if(descuento > descuentoMayor)
            descuentoMayor = descuento;
      }
      return descuentoMayor;
}
```

5. RECOMENDACIONES

Como conclusión general, se demuestra que aplicando principios y patrones de diseño podemos lograr desarrollar aplicaciones flexibles a los cambios. Por lo tanto, recomendamos el uso de estos principios y patrones cuando se desee garantizar una alta flexibilidad en el software, especialmente para tipos de software que manejen muchas reglas de negocio y que puedan ir cambiando en el tiempo. También se recomienda el uso de algún framework que permita la creación de clases, es decir, que haga las veces de las clases fábrica que se ha descrito en este artículo, por ejemplo se podría usar Spring Framework.

6. REFERENCIAS BIBLIOGRÁFICAS

[1] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sornmerla, Michael Stal. "PATTERN - ORIENTED SOFTWARE ARCHITECTURE, Volumen 1: A System of Patterns". John Wiley & Sons, 1996.

[2] Len Bass, Paul Clements, Rick Kazman. "Software Architecture in Practice". Third Edition, Addison Wesley, 2013.

[3] Erich Gamma. "Patrones de Diseño". Addison Wesley, 1995.

[4] Craig Larman. "UML Y PATRONES, Una introducción al análisis y diseño orientado a