**Documento Formal**

**Nombre: Daniel Fernando Santos Bustos**

**Fecha: 13 – Oct -2017**

**¿Cuáles son los requerimientos de la aplicación?**

1. Separar la creación de ordenes de la liquidación de órdenes.

**¿Qué se desea obtener?**

Se desea modificar alguna orden para calcular el monto total de la orden sin incurrir al ingreso de todas las ordenes de nuevo.

**¿Cuáles son los problemas actuales de la aplicación?**

1. No permite actualizar ni eliminar ordenes en especifico
2. Hay graves problemas de acoplamiento
   1. Módulos de alto nivel como la clase visitante dependen de módulos de bajo nivel como lo es la interfaz grafica
   2. Se asignan más de una responsabilidad a diferentes entidades, en especifico
      1. La interfaz grafica
      2. El visitante de ordenes

**¿Qué cambios se realizaron?**

1. Se creo una nueva interfaz gráfica que a la vista de diferentes personas tiene una mejor experiencia de usuario
   1. Se crearon las opciones de
      1. Actualización
      2. Eliminación
      3. Consulta
2. La liquidación de cada orden se realiza sobre una colección de ordenes
3. Se aplicaron buenas prácticas para mejorar la mantenibilidad y extensión del código.

**¿Cuáles patrones adicionales se aplicaron para resolver la problemática?**

* 1. ***Factory Method:*** Dado el tipo de orden seleccionado en el JCombobox se crea la opción de usuario, esta responsabilidad no debe ser delegada a la interfaz por los siguientes principios.
     1. Open-Closed : Si queremos crear una nueva orden tendremos que ir a modificar el código de la interfaz lo que causaría un alto acoplamiento
     2. Inversión de dependencias: Una operación de creación no debe depender de la interfaz gráfica para poder ejecutarse. “Módulos de alto nivel no deben depender de módulos de bajo nivel.”
     3. Responsabilidad Única: La interfaz grafica solo debería ser responsable en lidiar con tareas concernientes a la interfaz grafica.

**¿Dónde se ve reflejado el Factory method?**

* + 1. En la clase ***OrderFactory***
  1. ***Iterator***: Debido a que por naturaleza el patrón visitante es un patrón que trabaja sobre una colección, se almacenaron las ordenes en una colección, en una primera versión esta colección estaba en el visitante, donde a medida que se creaba una orden se almacenaba y cuando se pedía calcular el total la clase visitante recorría esta colección y calculaba la liquidación de cada orden que iba acumulando en una variable. Evidentemente se le está delegando dos responsabilidades calcular la liquidación “visitar” y realizar operaciones sobre la colección como por ejemplo calcular el monto total, eliminar un elemento de la colección, actualizar un elemento de la colección. Así que la solución fue separar esta colección en una clase que se le delegara esta responsabilidad.

**¿En cuál clase se aplicó el patrón?**

* En la clase ***RegisteredOrders***

**¿Qué enfoque de iterador se utilizó?**

* Pareciera que se utilizo el enfoque de filtrado, pero en este caso no, ya que la colección original la tiene la clase ***RegisteredOrders***
  1. ***Singleton***: Se utilizo inicialización temprana

**¿Cuántas ejemplificaciones son necesarias para una fábrica?**

En este caso solo hay necesidad de tener una fábrica creadora que provea todas las ejemplificaciones que se necesitan, para hacerlo más escalable se podría crear un pool de factorías que permitan crear distintos objetos.

**¿Cuántos iteradores se necesitan en la aplicación?. Para este problema en específico ¿Es necesario tener más de dos iteradores?**

En este problema se recorre la colección de una única forma y sobre cada elemento de la colección se calcula una operación, solo hay necesidad de una, ahora como la clase iterador tiene la colección de elementos ¿Qué pasaría si tuviéramos dos clases iterador?, podríamos calcular el monto total mal lo que generaría inconsistencias en la liquidación total.

**¿En qué clases se aplicó el patrón singleton?**

* RegisteredOrders
* OrderFactory
  1. Prototype: Dado a que en la especificación del problema en el libro Kuchana se menciona que existen muchas ordenes se podría inferir necesario el pratron prototype en este caso va a contener tantas ordenes como tipo de ordenes se tengan,

**¿Cuál tipo de copia se utilizó en el prototype?**

Se utilizo copia superficial ya que las ordenes no tienen referencias a objetos que son las que persisten cuando se copia un objeto por copia superficial.

**¿En cual clase se utilizó Prototype?**

Ya que esta ligado con la creación este patrón se utilizó en la factoría de ordenes ya que es la responsable de obtener las ordenes.