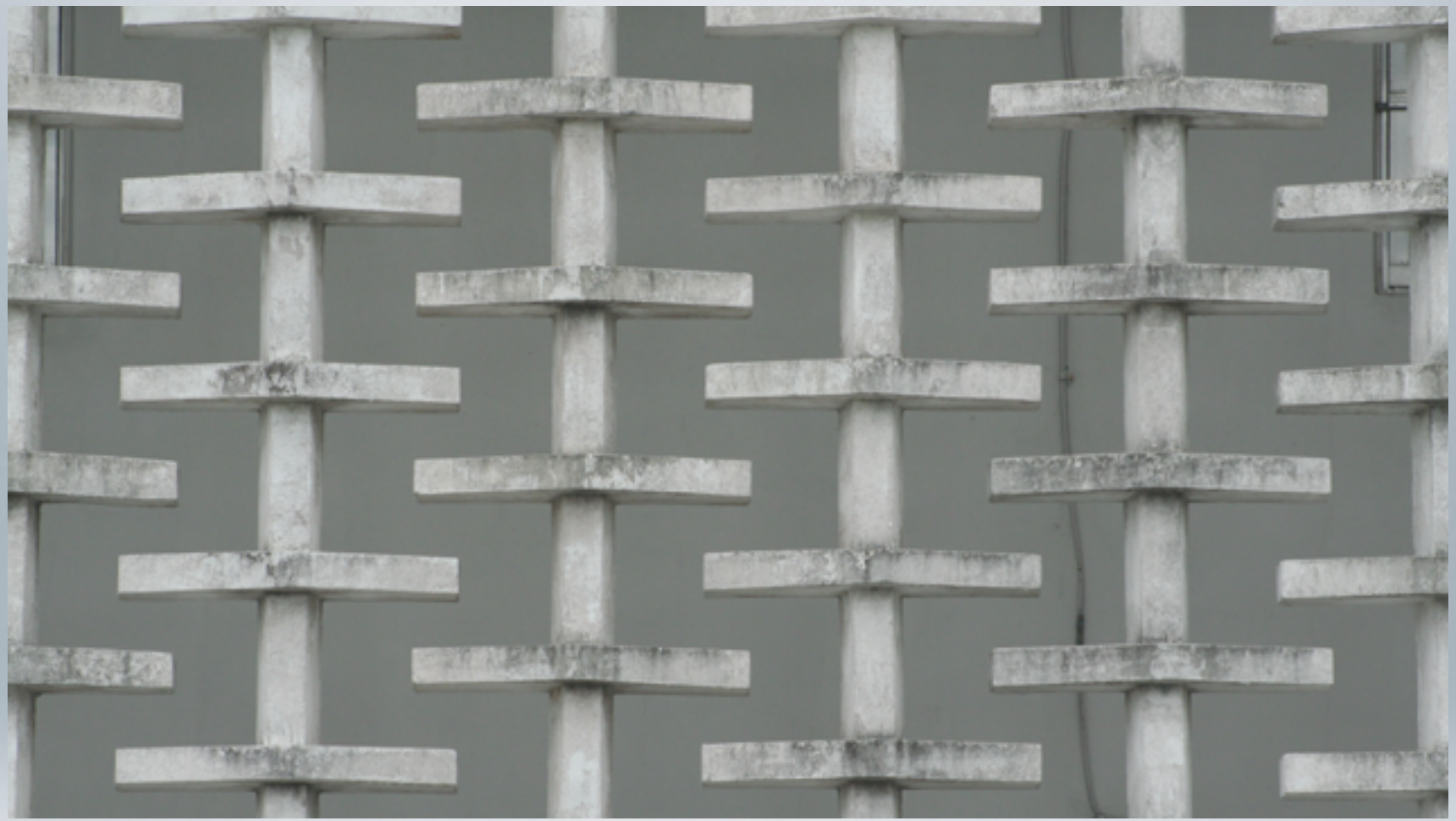


Patrones de Diseño

Analista Programador

Francisco Villegas - www.franciscovillegas.net - 2013

Que es un patrón

- Es una descripción de un problema y su solución que recibe un nombre y que puede puede aplicar en nuevos contextos.
- Los patrones tienen nombres sugerentes que ayudan a identificarlos y facilitan la comunicación.
- Una adecuada asignación de responsabilidades entre objetos contribuye al desarrollo de sistemas

Notación

- **Nombre del Patrón:** Permite describir en pocas palabras un problema de diseño junto con sus solución y consecuencias
- **Problema que resuelve:** describe cuando aplicar el patrón. Explica el problema y el contexto.
- **Solución:** elementos que constituyen el diseño, sus relaciones, responsabilidades y colaboraciones.

GRASP

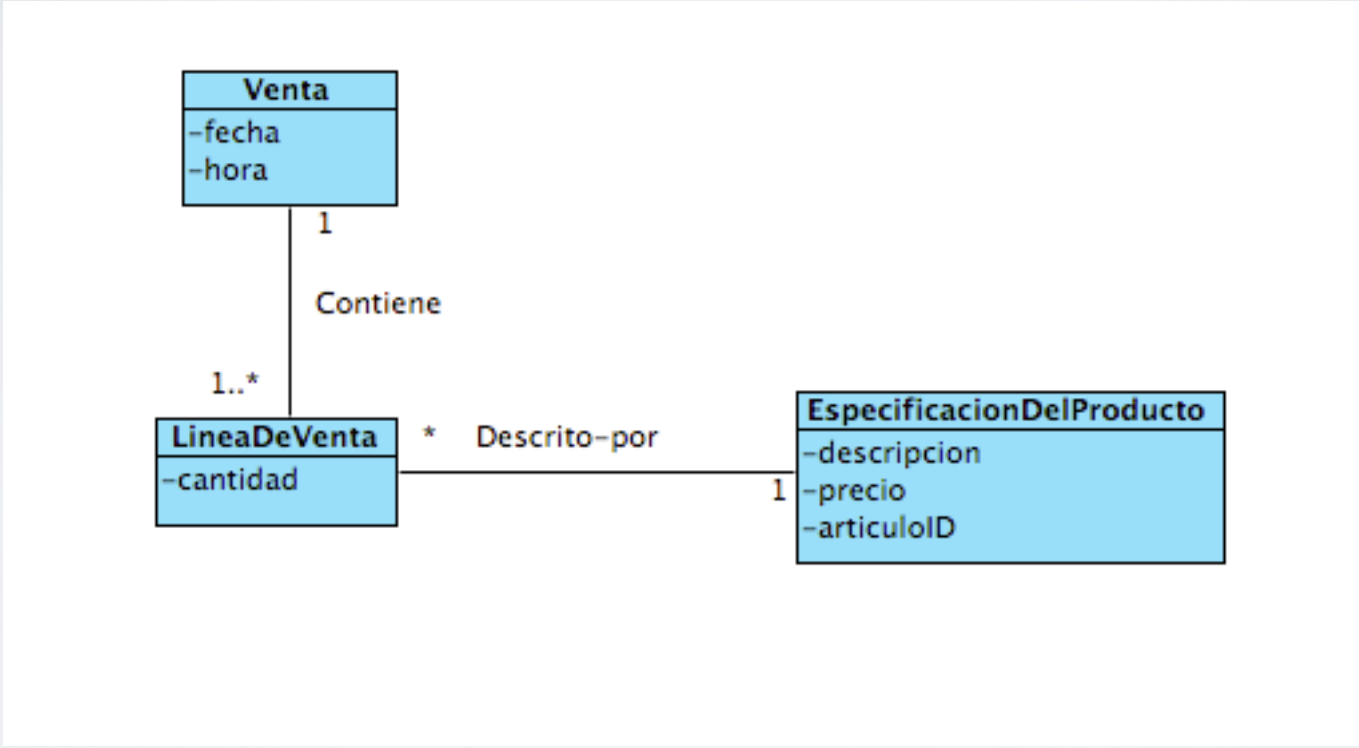
General Responsibility Assignment Software Patterns (or Principles)

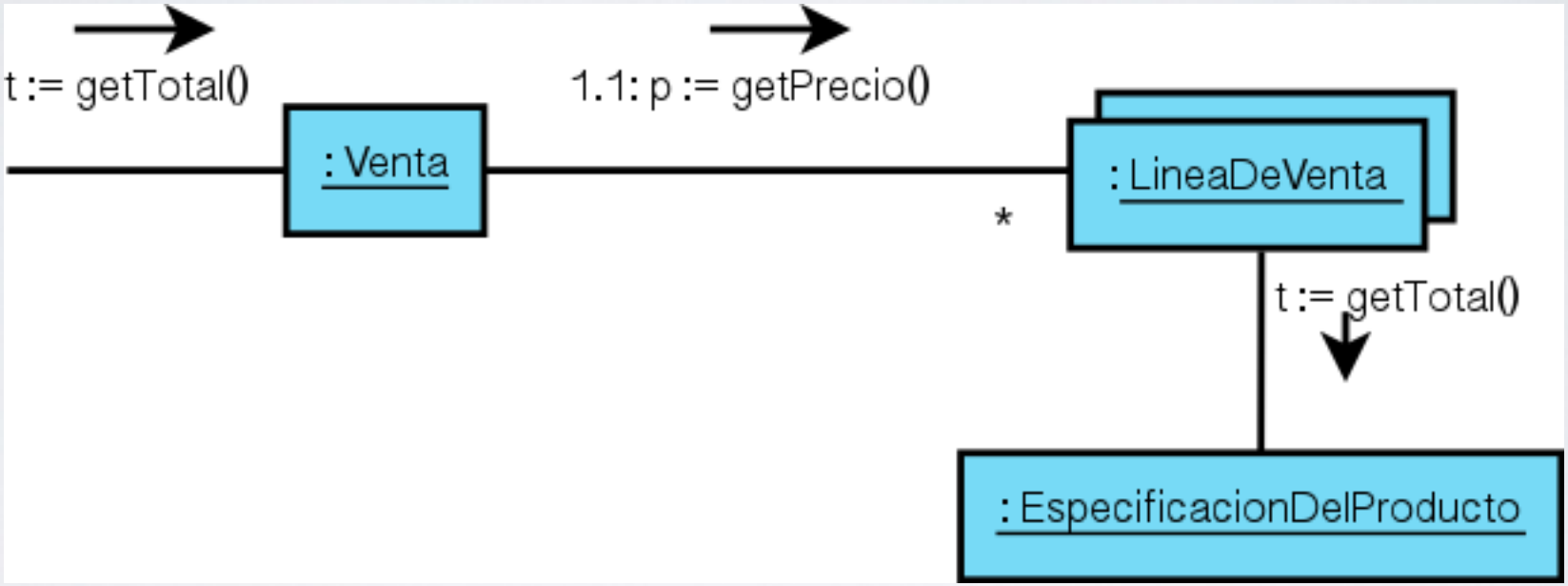
- Describen los principios fundamentales del diseño de objetos y la asignación de responsabilidades, expresados como patrones.
  - Experto
  - Creador
  - Alta Cohesión
  - Bajo Acoplamiento
  - Controlador

Experto

- **Nombre:** Experto
- **Problema:** ¿Cuál es un principio general para asignar responsabilidades a los objetos?
- **Solución:** Asignar una responsabilidad al experto en información -la clase que tiene la información necesaria para realizar la responsabilidad-

Ejemplo Experto




t := getTotal()
: Venta
1.1: p := getPrecio()
*
: LineaDeVenta
t := getTotal()
: EspecificacionDelProducto

Beneficios

- Se mantiene el encapsulamiento de la información, puesto que los objetos utilizan su propia información para llevar a cabo las tareas. Generalmente estimula el **Bajo acoplamiento** entre las clases.

- Se distribuye el comportamiento entre las clases que contienen la información requerida, por tanto, se estimula loas definiciones de clases más cohesivas y "ligeras" que son más fáciles de entender y mantener. Se soporta normalmente **Alta cohesión**.

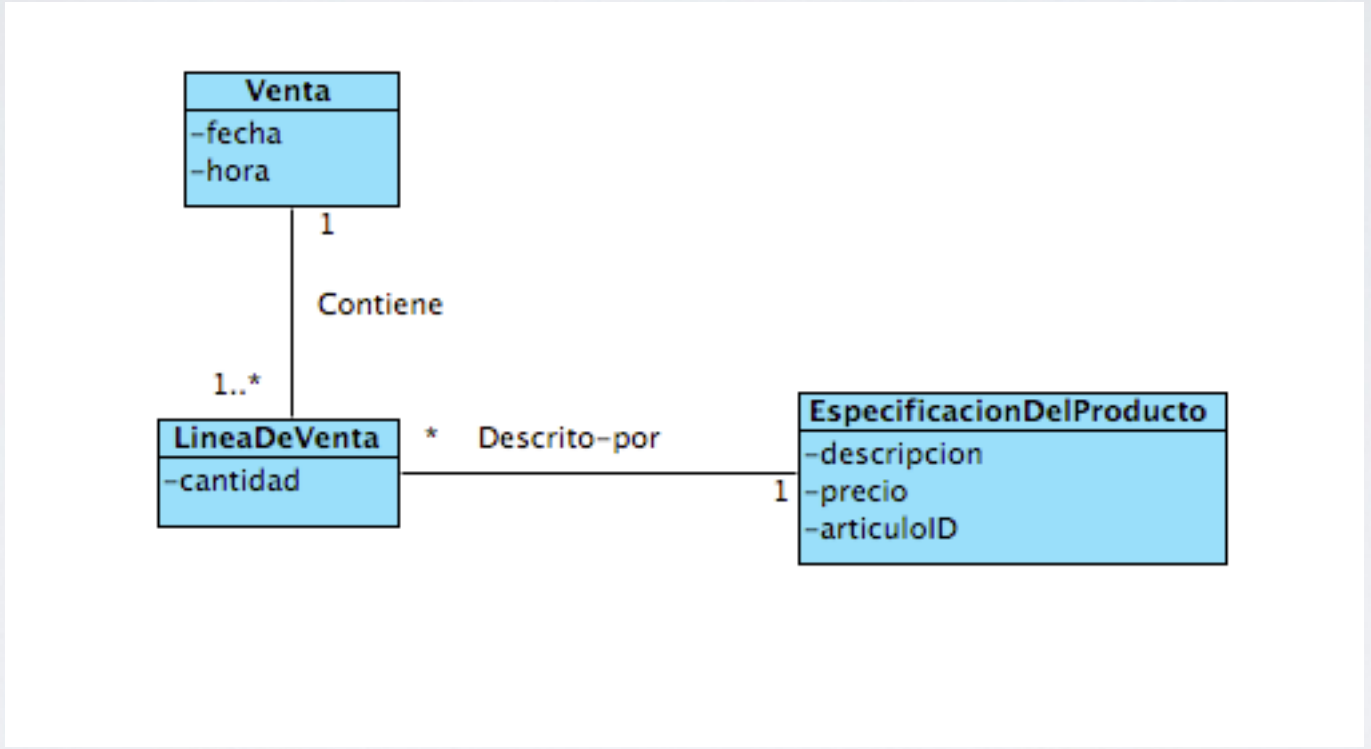
Creador

- Nombre: Creador
- Problema: ¿Quién debería ser el responsable de la creación de una nueva instancia de alguna clase?
- Solución: Asignar a la clase B la responsabilidad de crear una instancia de clase A si se cumple uno o más de los casos siguientes:

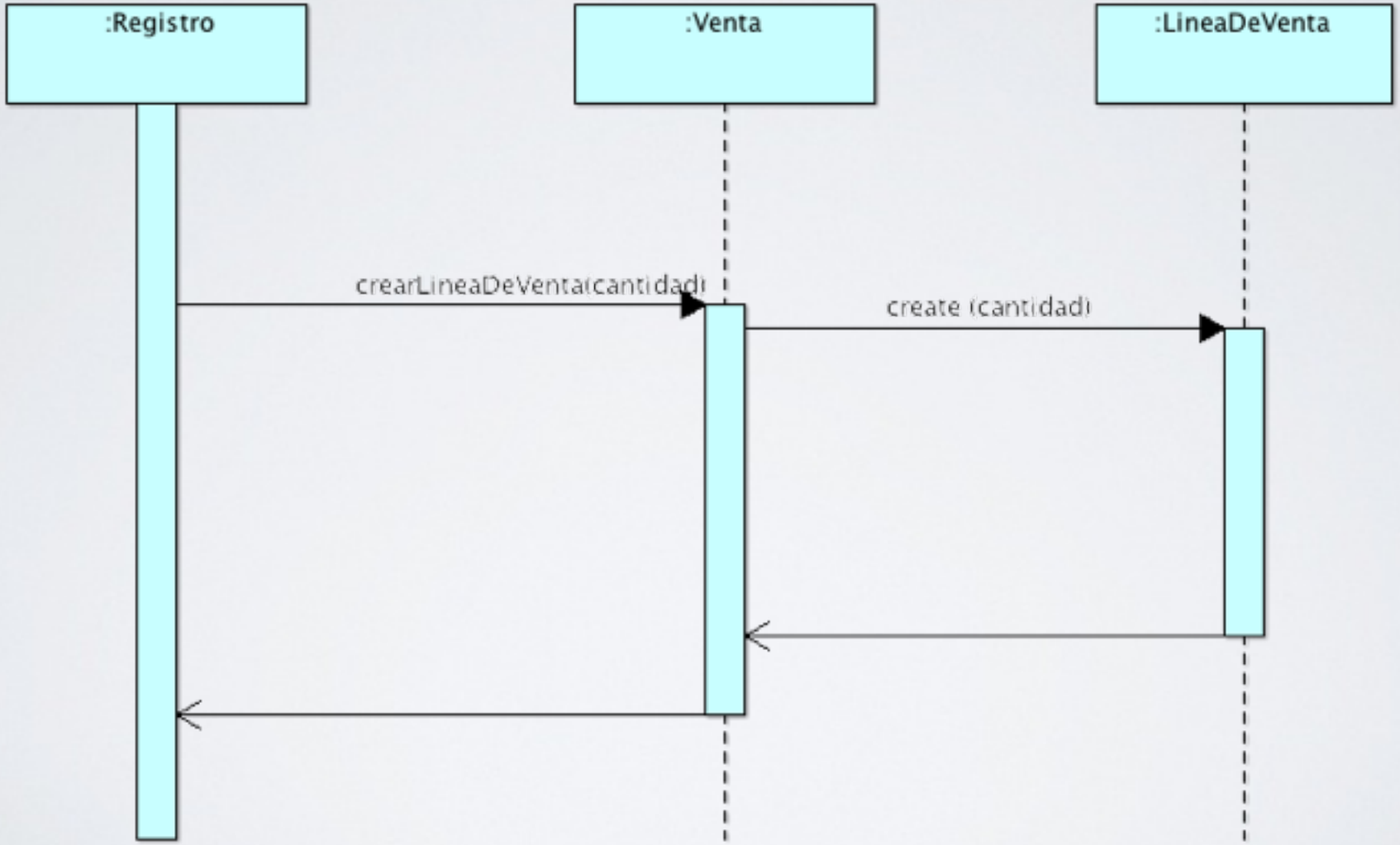
Creador (cont.)

- B agrega objetos de A
- B contiene objetos de A
- B registra instancias de objetos de A
- B utiliza más estrechamente objetos de A
- B tiene los datos de inicialización que se pasarán a un objeto de A cuando sea creado. Por lo tanto, B es un **Experto** con respecto a la creación de A.

Ejemplo Creador



Ejemplo Creador



Alta Cohesión

- **Nombre:** Alta Cohesión
- **Problema:** ¿Cómo mantener la complejidad manejable?
- **Solución:** Asignar una responsabilidad de manera que la cohesión permanezca alta.

Cohesión: Acción y efecto de reunirse o adherirse las cosas entre sí o la materia de que están formadas.

Alta Cohesión

- **Baja Cohesión:** una clase tiene la responsabilidad exclusiva de una tarea compleja dentro de un área funcional.
- **Alta Cohesión:** Una clase tiene responsabilidades moderadas en un área funcional y colabora con otras para realizar las tareas.

Alta Cohesión : Beneficios

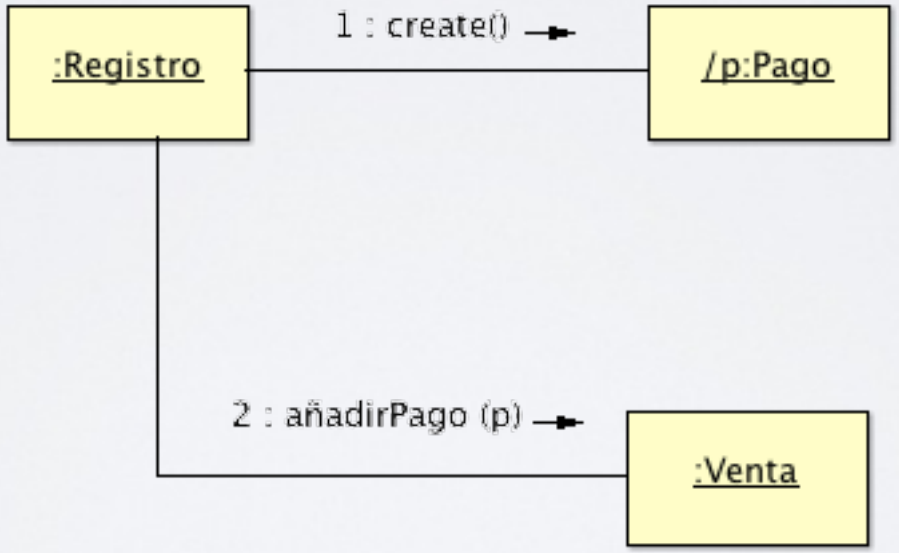
- Se incrementa la claridad y facilita la compresión del diseño.
- Se facilita el mantenimiento y mejoras
- Se incrementa el **Bajo Acoplamiento**.
- Se incrementa la reutilización de las clases.

Bajo Acoplamiento

- **Nombre:** Bajo Acoplamiento
- **Problema:** ¿Cómo soportar bajas dependencias, bajo impacto del cambio e incremento de la reutilización?
- **Solución:** Asignar una responsabilidad de manera que el acoplamiento permanezca bajo.

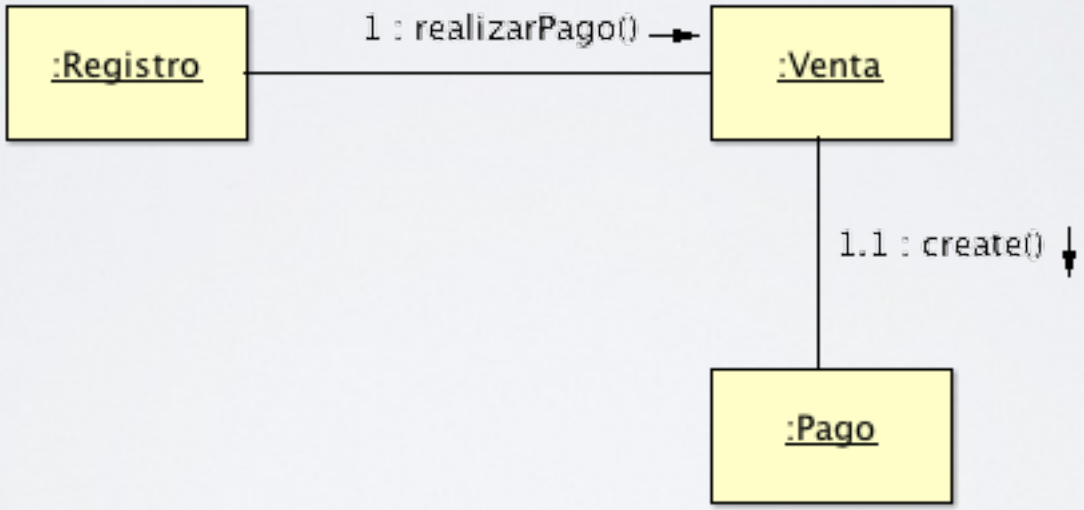
Bajo Acoplamiento : Ejemplo

Antes



Bajo Acoplamiento : Ejemplo

Después



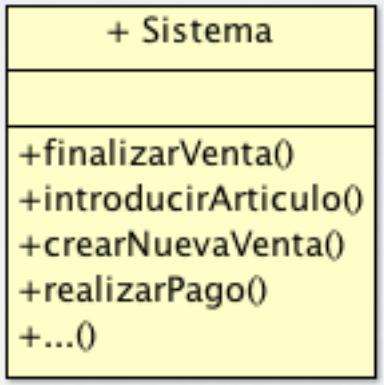
Bajo Acoplamiento : Beneficios

- No afectan los cambios en otros componentes
- Fácil de entender de manera aislada
- Conveniente para reutilizar

Controlador

- Nombre: Controlador
- Problema: ¿Quién debe ser el responsable de gestionar un evneto de entrada al sistema?
- Solución: Asignar la responsabilidad de recibir o manejar un mensaje de evento del sistema a una clase que representa una de las siguientes opciones:
  - Representa el sistema global, dispositivo o subsistema (**Controlador de fachada**)
  - Representa un escenario de caso de uso en el que tiene lugar el evento del sistema. <NombreCU>Controlador (**Controlador de caso de uso**)

controlador : Ejemplo



Controlador : Ventajas

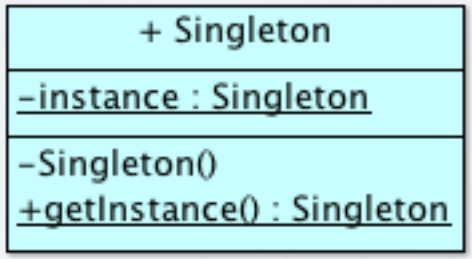
Aumenta el potencial para reutilizar las interfaces

Razonamiento sobre el estado (secuencia de pasos) de los casos de uso.

Singleton

- **Nombre:** Singleton
- **Problema:** Se quiere dada una clase, solo exista una instancia de la misma.
- **Solución:**

Structura Singleton



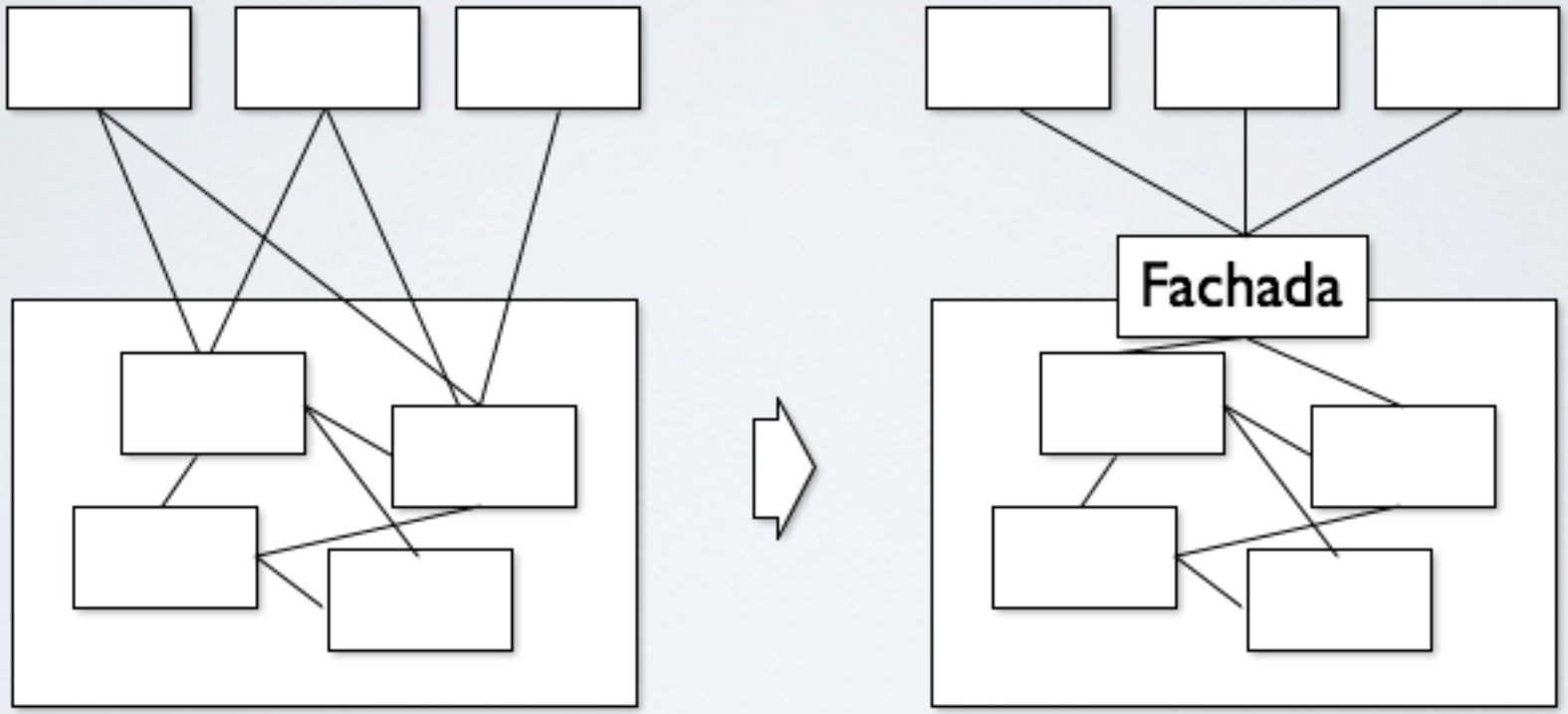
Singleton : Ejemplo

```
public class Singleton {

        private static Singleton instance;

        public static synchronized Singleton getInstance() {

                if(instance == null) {
                        instance = new Singleton();
                }

                return instance;
        }

        private Singleton() {  }

        ...
}
```

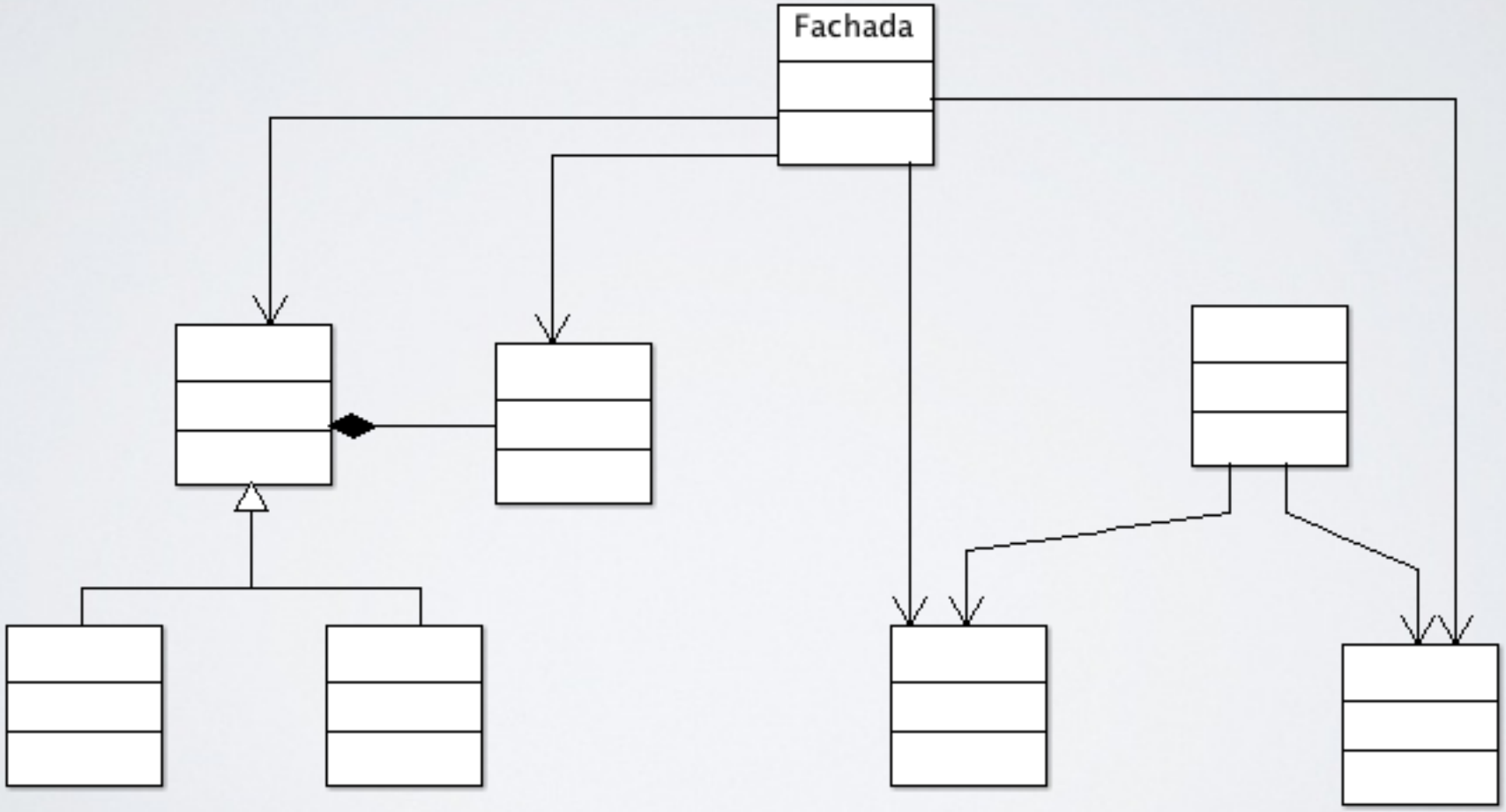
Facade

- **Nombre:** Facade
- **Propósito:** Proporciona una interfaz unificada para un conjunto de interfaces de un subsistema. Define una interfaz de alto nivel que hace que el subsistema sea más fácil de usar.
- **Solución:**

Facade



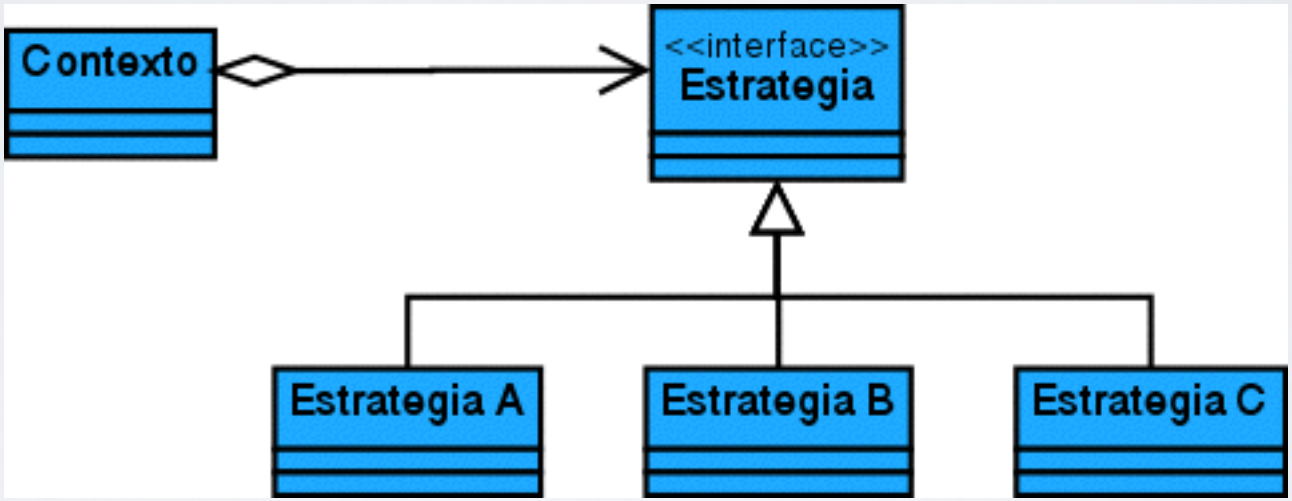
Facade : Estructura



Strategy (comportamiento)

Define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables. Permite que un algoritmo varíe independientemente de los clientes que lo usan.

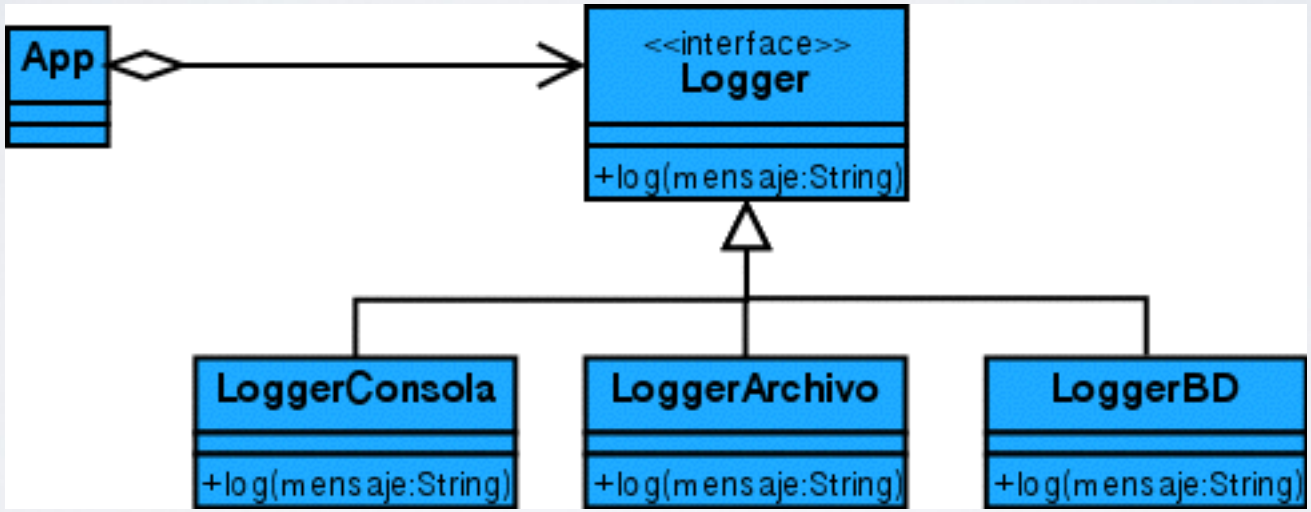
Estructura


Contexto
<<interface>>
Estrategia
Estrategia A
Estrategia B
Estrategia C

Aplicabilidad

- Muchas clases relacionadas difieren sólo en su comportamiento
- Variantes de un algoritmo
- Un algoritmo usa datos que los cliente no deberían conocer
- Una clase define muchos comportamientos

Ejemplo

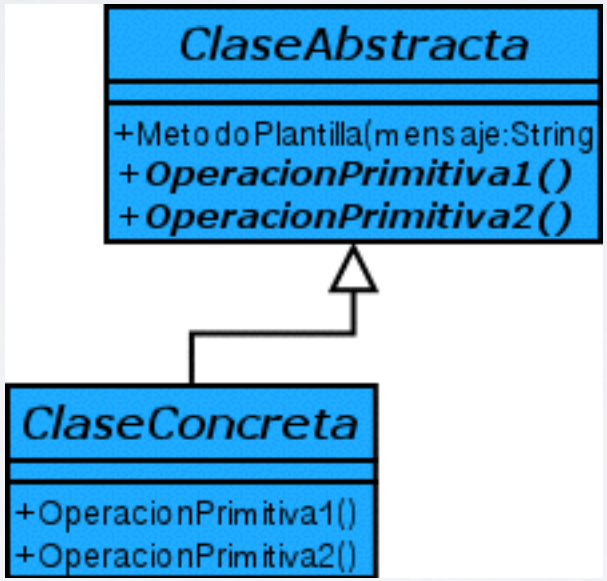

App
<<interface>>
Logger
+log(mensaje:String)
LoggerConsola
+log(mensaje:String)
LoggerArchivo
+log(mensaje:String)
LoggerBD
+log(mensaje:String)

Template

Define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos.

Permite que las subclases redefinan ciertos pasos de un algoritmo sin cambiar su estructura.

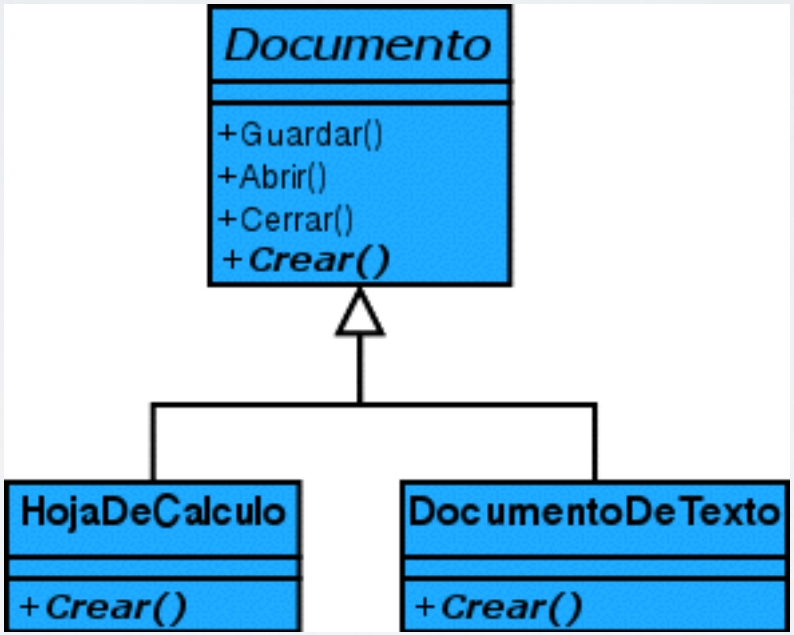
Estructura


ClaseAbstracta
+MetodoPlantilla(mensaje:String
+OperacionPrimitiva1()
+OperacionPrimitiva2()
ClaseConcreta
+OperacionPrimitiva1()
+OperacionPrimitiva2()

Aplicabilidad

- Para implementar las partes de un algoritmo que no cambian y dejar que sean las subclases quienes implementen el comportamiento que pueden variar.
- cuando el comportamiento repetido de varias subclases debería refactorizarse y ser localizado en un clase común para evitar código duplicado.
- para controlar las extensiones de las subclases.

Ejemplo


Documento
+Guardar()
+Abrir()
+Cerrar()
+Crear()
HojaDeCalculo
+Crear()
DocumentoDeTexto
+Crear()

Referencias

- Craig Larman. UML y Patrones. 2° edición. ISBN: 84-205-3438-2
- Gamma, E.; Helm, R; Johnson, R.; Vlissides, J. Patrones de Diseño. ISBN: 84-7829-059-1