

Diseño de Sistemas

Unidad 3: Diseño con Objetos - parte 2

Martín Agüero



UTN.BA

DPTO. INGENIERÍA EN SISTEMAS DE INFORMACIÓN
CÁTEDRA DISEÑO DE SISTEMAS

A wide-angle photograph of a lush green field of grain, likely sorghum, stretching towards a horizon under a warm, golden sunset sky. The sun is low on the horizon, creating a strong glow and long, soft shadows across the field. The sky transitions from a pale yellow near the horizon to a soft orange and then a pale blue at the top.

Unidad 3:

*Patrones Creacionales. Patrones estructurales. Patrones de comportamiento.
Refactorización mediante Patrones.*

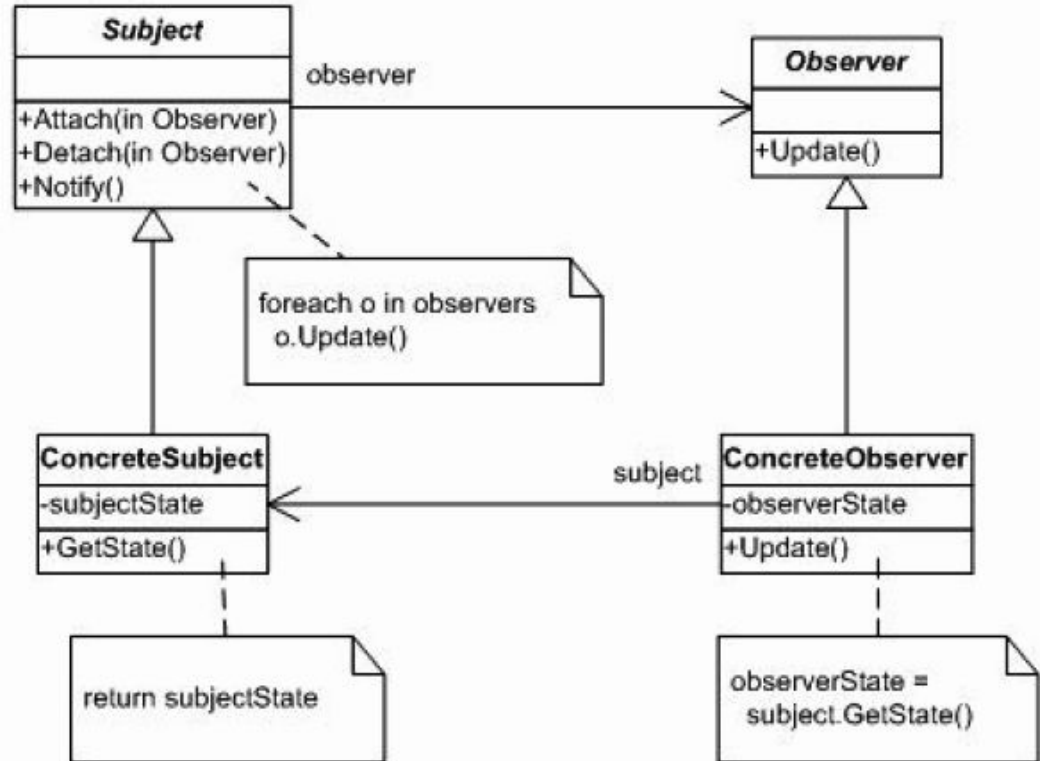
Patrones de Diseño

Observer

Tipo: Comportamiento

Propósito:

Definir dependencias one-to-many entre objetos, de forma tal que cuando un objeto cambia su estado todos los objetos dependientes son notificados y actualizados inmediatamente.



Patrones de Diseño

Observer

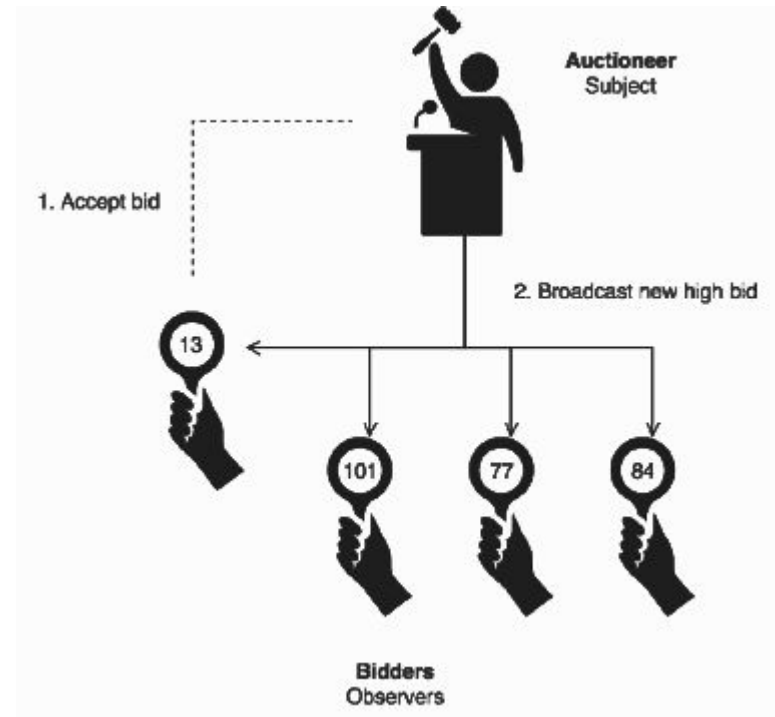
Beneficios:

👍 Ayuda a desacoplar

👍 Mejora la performance, ya que el observer no tiene que hacer poll para verificar si hay cambios.

Ejemplo:

Cada oferente posee una paleta numerada que se utiliza para indicar una oferta. El subastador comienza la licitación, y "observa" cuando se eleva una paleta para aceptar la oferta. La aceptación de la oferta cambia el precio de la oferta que se emite a todos los compradores.



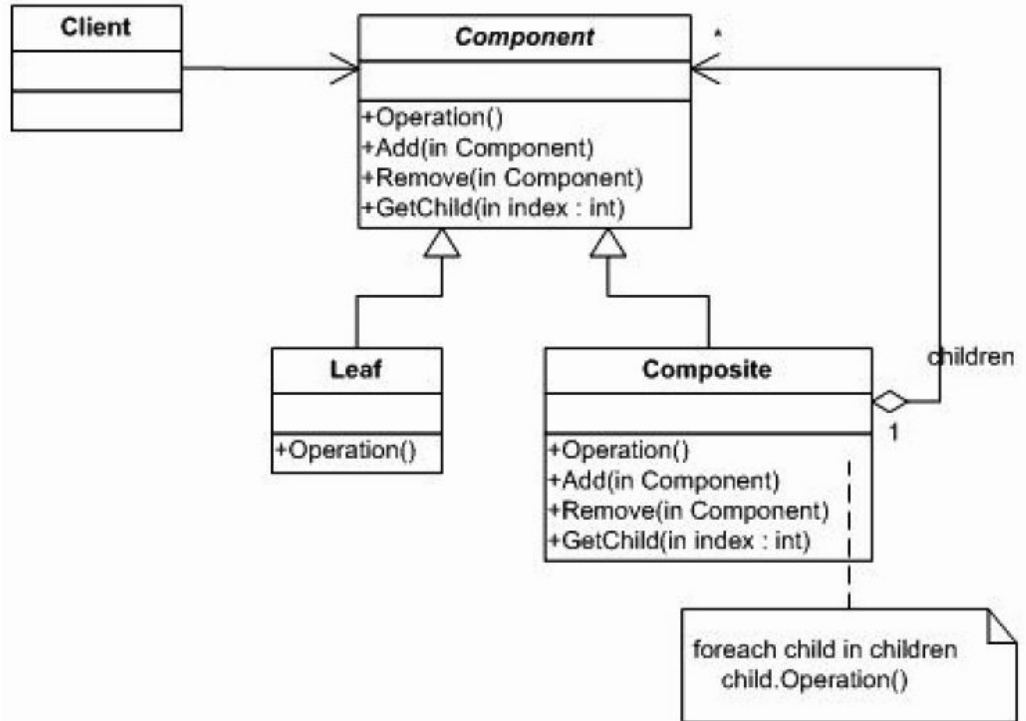
Patrones de Diseño

Composite

Tipo: Estructural

Propósito:

Componer objetos en estructuras de árbol para representar jerarquías. Permite tratar objetos individuales o compuestos de la misma manera.



Patrones de Diseño

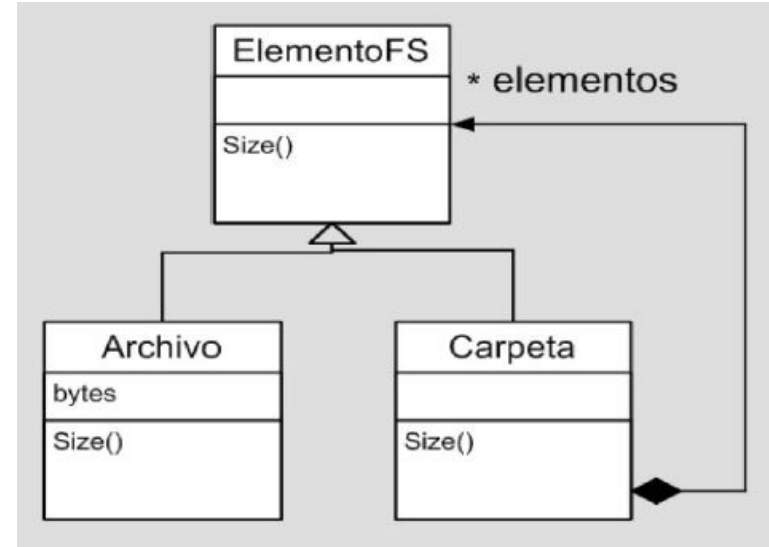
Composite

Desventaja:

👎 Un Component puede aceptar cualquier otro Component (cualquier otra subclase o implementación de Component) debido a la herencia. Hay que hacer validaciones si es necesario.

Ejemplo:

Sistema de archivos del sistema operativo.



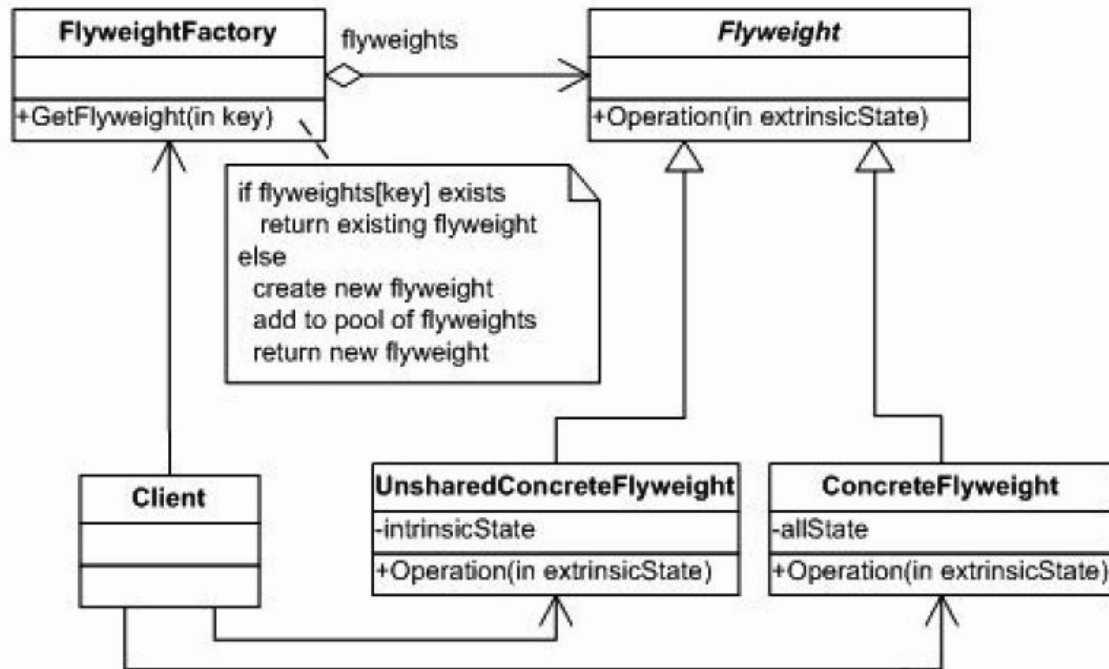
Patrones de Diseño

Flyweight

Tipo: Estructural

Propósito:

Emplear objetos compartidos que son similares, en lugar de crear nuevas instancias. Se usa cuando se necesita crear muchos objetos similares. Mejora la performance, ya que reduce el uso de memoria.



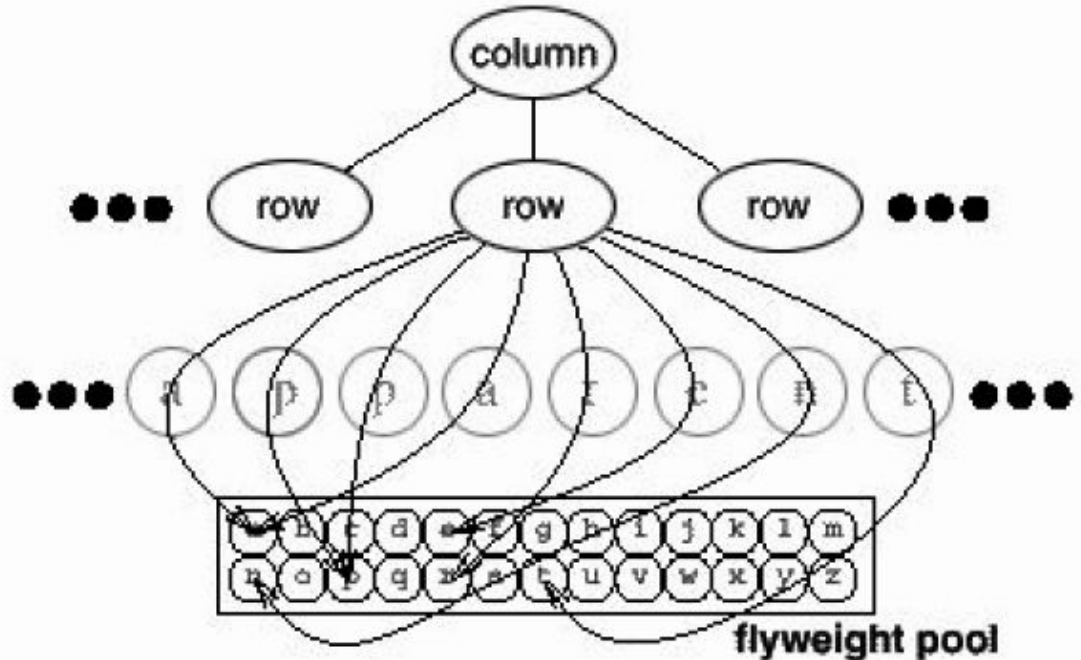
Factory almacena una única instancia (o un pool acotado de instancias) de los **ConcreteFlyweight** para un key determinado. Los **ConcreteFlyweight** debería ser stateless (sin estado) ya que se pueden utilizar en diferentes contextos.

Patrones de Diseño

Flyweight

Ejemplo:

En un editor de texto, en lugar de repetir la instancia a cada símbolo, se define una referencia a ese en el flyweight pool.



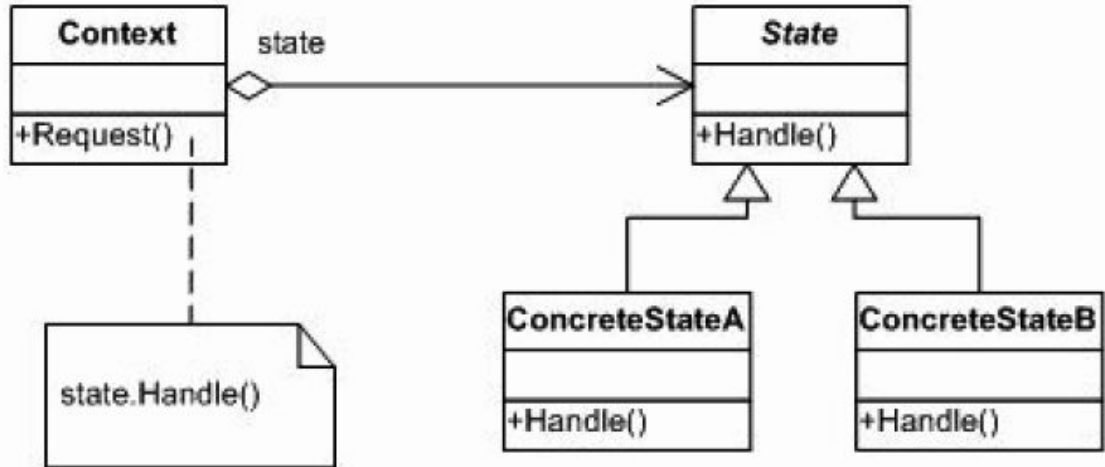
Patrones de Diseño

State

Tipo: Comportamiento

Propósito:

Permitir que un objeto altere su comportamiento cuando su estado interno cambia. Permite modelar las transiciones entre estados.



Patrones de Diseño

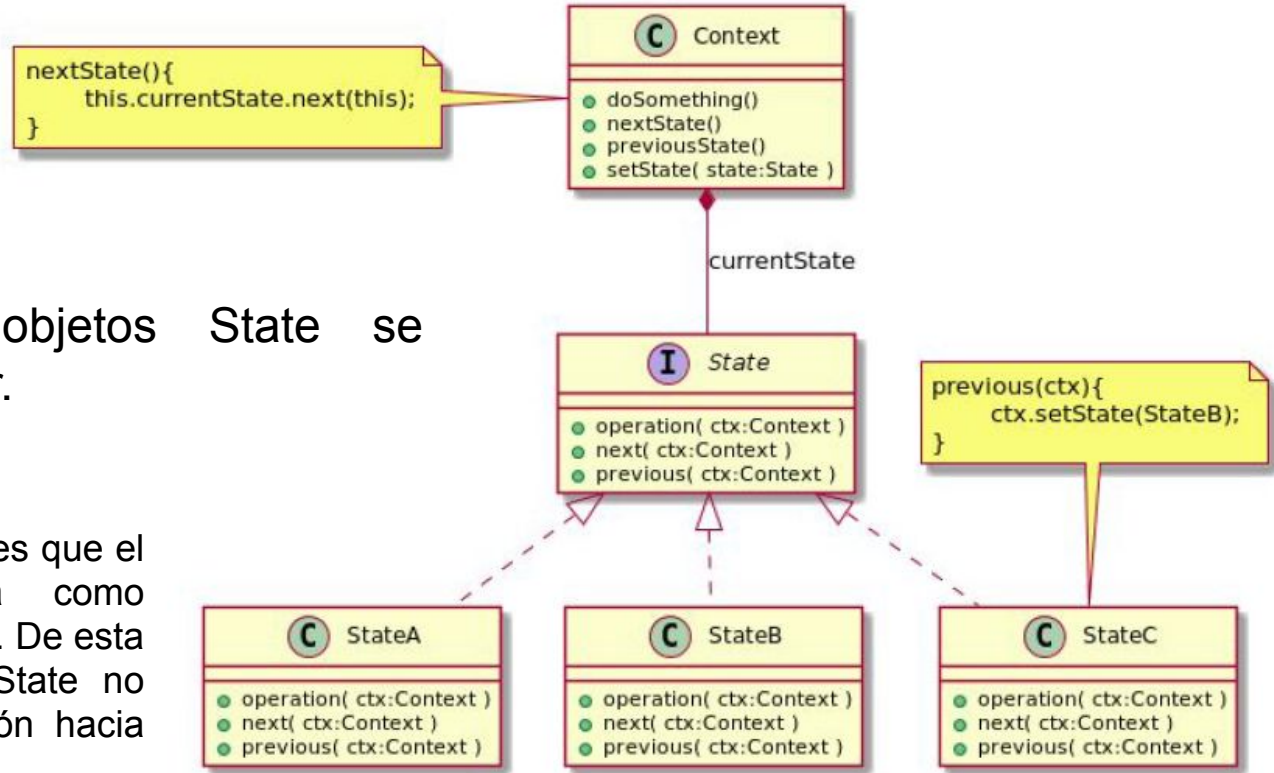
State

Beneficios:

- 👍 Desacoplamiento
- 👍 Mantenimiento
- 👍 Instancias de objetos State se pueden compartir.

Ejemplo:

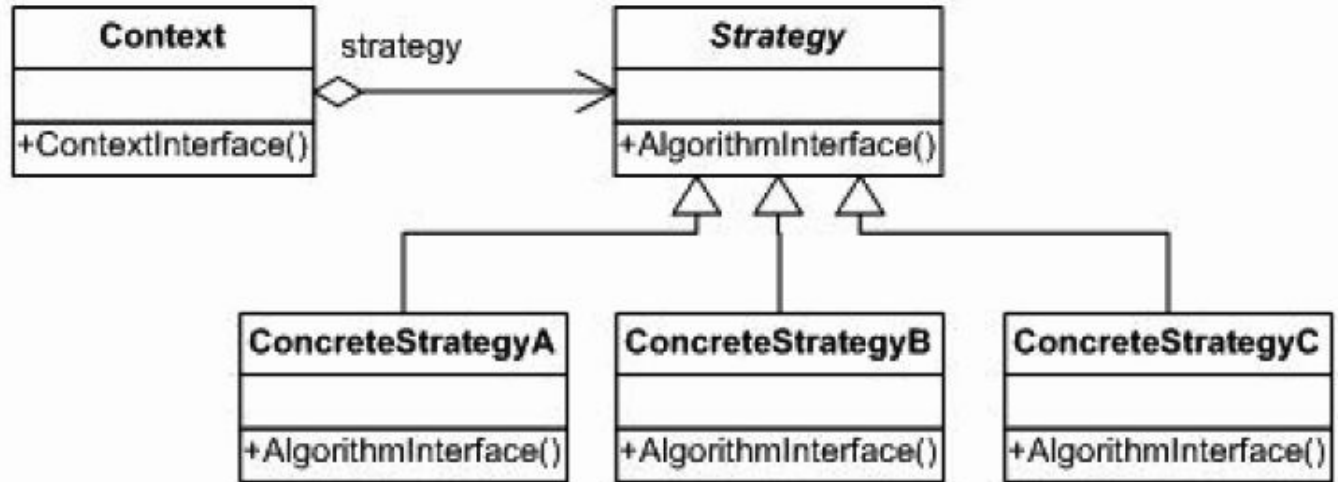
Una variante muy utilizada es que el método `Handle()` reciba como parámetro el objeto `Context`. De esta forma, las instancias de `State` no necesitan tener una relación hacia `Context`.



Patrones de Diseño

Strategy

Tipo: Creacionales

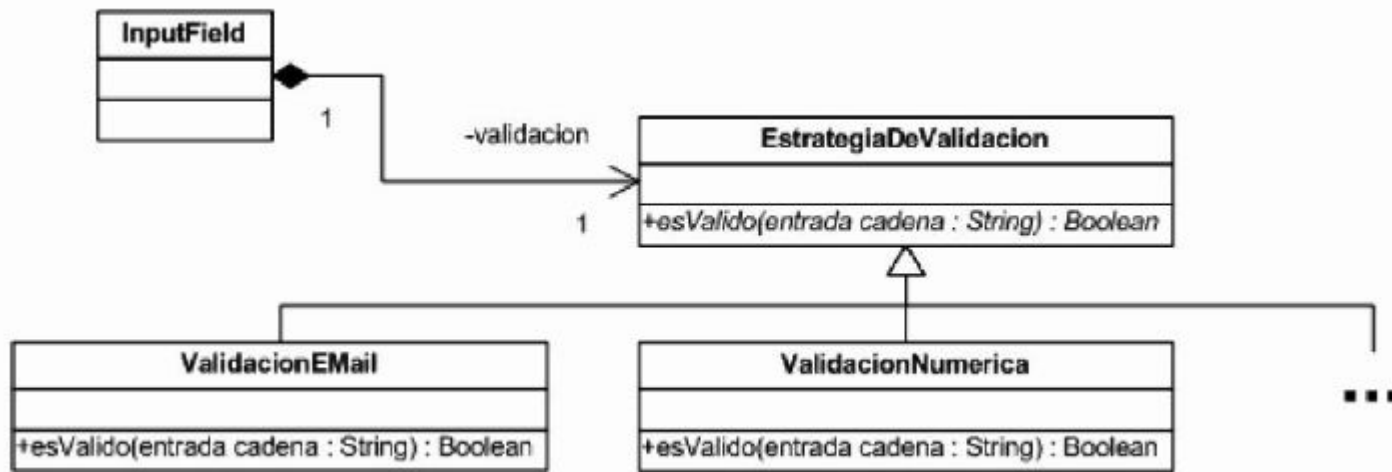


Propósito: Definir una familia de algoritmos, encapsularlos y hacerlos intercambiables. Permite a las clases cliente cambiar el algoritmo en tiempo de ejecución.

Patrones de Diseño

Strategy

Ejemplo:



Beneficios:

- 👍 Desacoplamiento (Las implementaciones concretas se pueden cambiar en forma dinámica/ Los clientes no conocen los detalles de implementación.)
- 👍 Mantenimiento
- 👍 Alternativa a la herencia (subclassing)
- 👍 Ayuda a reducir el uso de condicionales.

Patrones de Diseño

Adapter

Objetivo: Convertir la protocolo de una clase en otra, que es la que el objeto cliente espera.

Problema: Muchas veces, clases que fueron pensadas para ser reutilizadas no pueden aprovecharse porque su protocolo no es compatible con el protocolo específico del dominio de aplicación con el que se está trabajando.

Se desea utilizar una clase existente, cuyo protocolo no es compatible con el requerido.

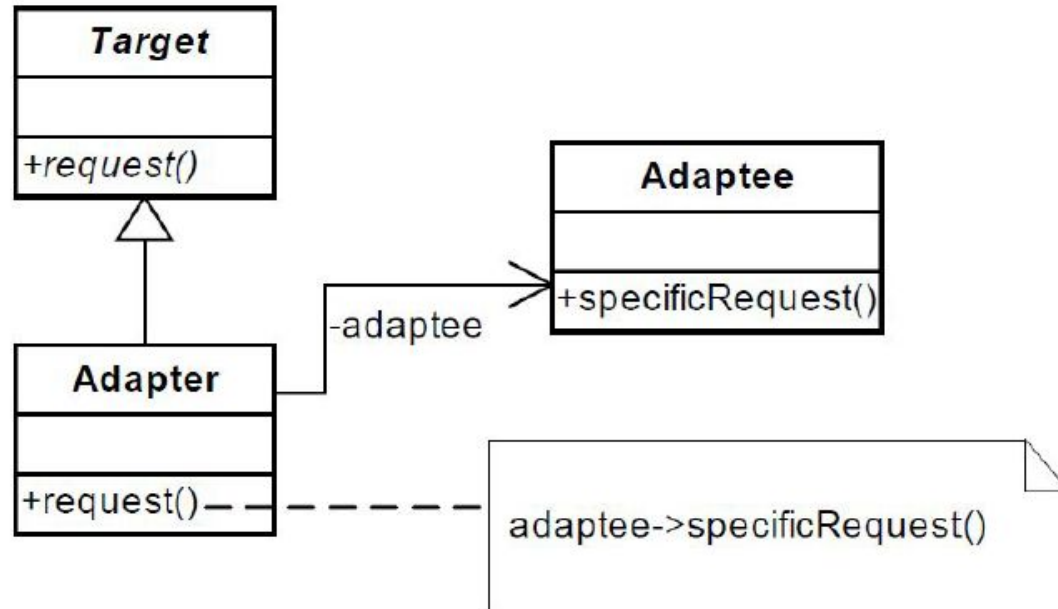
Se desea crear una clase que puede llegar a cooperar con otros objetos cuyo protocolo no se puede predecir de antemano.

Solución: Crear una clase que se encargue de “transformar” los nombres de los mensajes.

Patrones de Diseño

Adapter

Tipo: Creacionales



Propósito: Convertir la interfaz de una clase en otra interfaz que el cliente espera recibir. Permite que dos clases incompatibles puedan funcionar en conjunto.

Actividad Práctica:

Qué patrón de diseño consideraría para cada requerimiento.

Req. 1: *Si un curso es agendado para ser eliminado, todos los usuarios suscriptos a él deberán ser notificados de forma automática.*

Req. 2: *Se desea que la app altere su comportamiento de forma dinámica en función de la calidad del canal de comunicaciones (WIFI, EDGE y 3G). Proponer un diseño mantenible que permita agregar más alternativas (por ejemplo LTE) con cierta facilidad.*

Req. 3: *Cada asignación de un corto a una franja horaria emite un certificado de declaración jurada, éste debe ser exportable a formato PDF, Docx, XML, JSON u otro a definir.*

Req. 4: *Listado de artículos. Los datos de los artículos a mostrar en un listado (por categoría o como resultado de una búsqueda) pueden provenir de diferentes fuentes de datos (servicios web de proveedores). El objetivo es poder recorrer de manera uniforme la colección de artículos recuperada.*

Req. 5: *Antes de ser emitido, cada corto debe ser procesado por una serie de validadores que analizarán, de forma secuencial e independiente, diferentes cualidades, como por ejemplo: duración, resolución de la imagen, calidad del audio, compresión u otros criterios que puedan agregarse para futuras campañas.*



Cargar los resultados en: <https://b.socrative.com/login/student/> (Habitación/Room: UTN**DDS**)

Patrones de Diseño

Refactorización

Patrones de Diseño

Refactorización

Es el proceso de cambio de un sistema de software de manera tal que no altera el comportamiento externo mientras mejora su estructura interna. Es una manera disciplinada para limpiar código que minimiza las posibilidades de introducir errores. En esencia, cuando se realiza refactoring, lo que se está haciendo es mejorar el diseño.



Imagen: <https://elearning.industriallogic.com>

Patrones de Diseño

Refactorización

El proceso de refactorización implica:

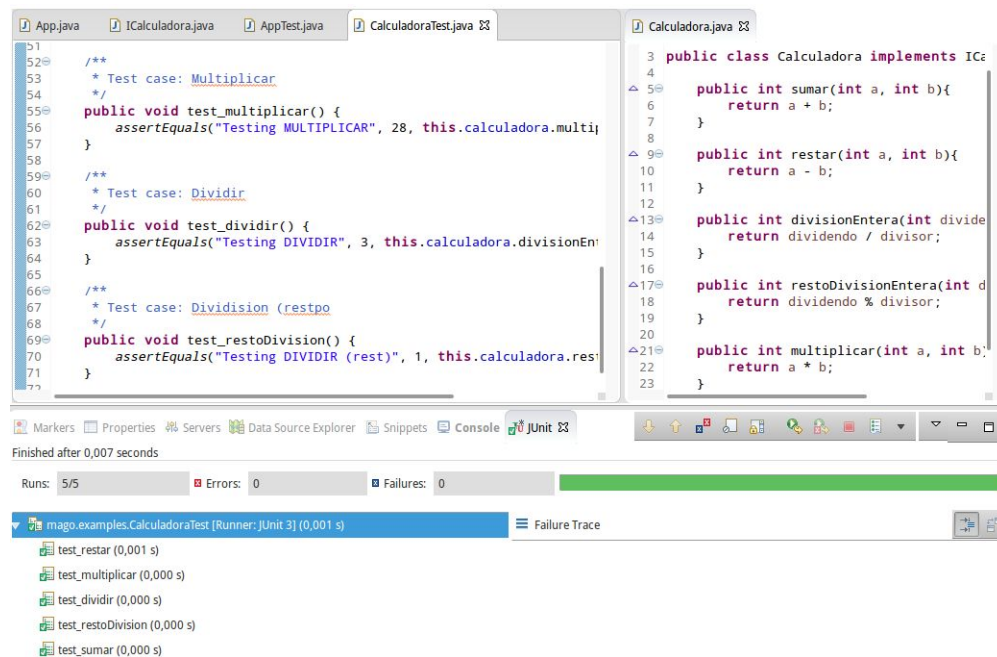
- ✓ Remover duplicados
- ✓ Simplificar lógica compleja
- ✓ Clarificar código confuso

Cuando se hace refactorización, se está optimizando el diseño del código fuente. Esas mejoras pueden ser algo pequeño como cambiar el nombre de una variable o importante como unificar dos jerarquías de clases.

Patrones de Diseño

Refactorización

Siempre que se realiza refactorización, el primer paso consiste en construir un sólido conjunto de pruebas unitarias para la sección de código fuente a modificar. Las pruebas son esenciales para contar con una referencia confiable del comportamiento o resultado esperado.



The screenshot displays an IDE with two open files: `Calculadora.java` and `CalculadoraTest.java`. The `Calculadora` class implements the `ICalculadora` interface with methods for sum, subtraction, division, and multiplication. The `CalculadoraTest` class contains four unit tests: `test_multiplicar`, `test_dividir`, `test_restoDivision`, and `test_restar`, each using `assertEquals` to verify the results. Below the code editor, the JUnit runner shows the test results, indicating that all tests passed successfully.

```
/**
 * Test case: Multiplicar
 */
public void test_multiplicar() {
    assertEquals("Testing MULTIPLICAR", 28, this.calculadora.multiplicar(4, 7));
}

/**
 * Test case: Dividir
 */
public void test_dividir() {
    assertEquals("Testing DIVIDIR", 3, this.calculadora.divisionEntera(12, 4));
}

/**
 * Test case: Division (resto)
 */
public void test_restoDivision() {
    assertEquals("Testing DIVIDIR (rest)", 1, this.calculadora.restoDivisionEntera(10, 3));
}

public class Calculadora implements ICalculadora {
    public int sumar(int a, int b){
        return a + b;
    }

    public int restar(int a, int b){
        return a - b;
    }

    public int divisionEntera(int dividendo, int divisor){
        return dividendo / divisor;
    }

    public int restoDivisionEntera(int dividendo, int divisor){
        return dividendo % divisor;
    }

    public int multiplicar(int a, int b){
        return a * b;
    }
}
```

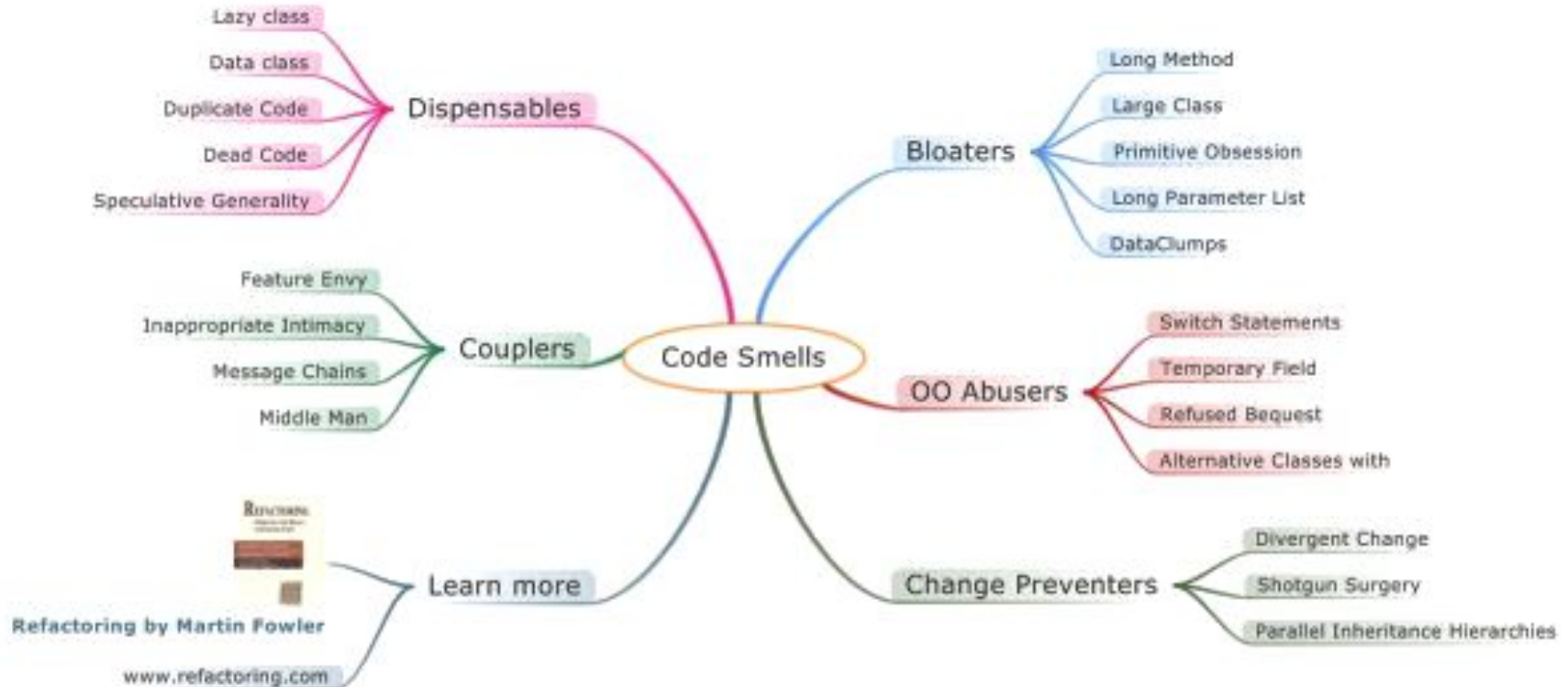
JUnit 3 [0,001 s]

- test_restar (0,001 s)
- test_multiplicar (0,000 s)
- test_dividir (0,000 s)
- test_restoDivision (0,000 s)
- test_sumar (0,000 s)

Patrones de Diseño

Refactorización

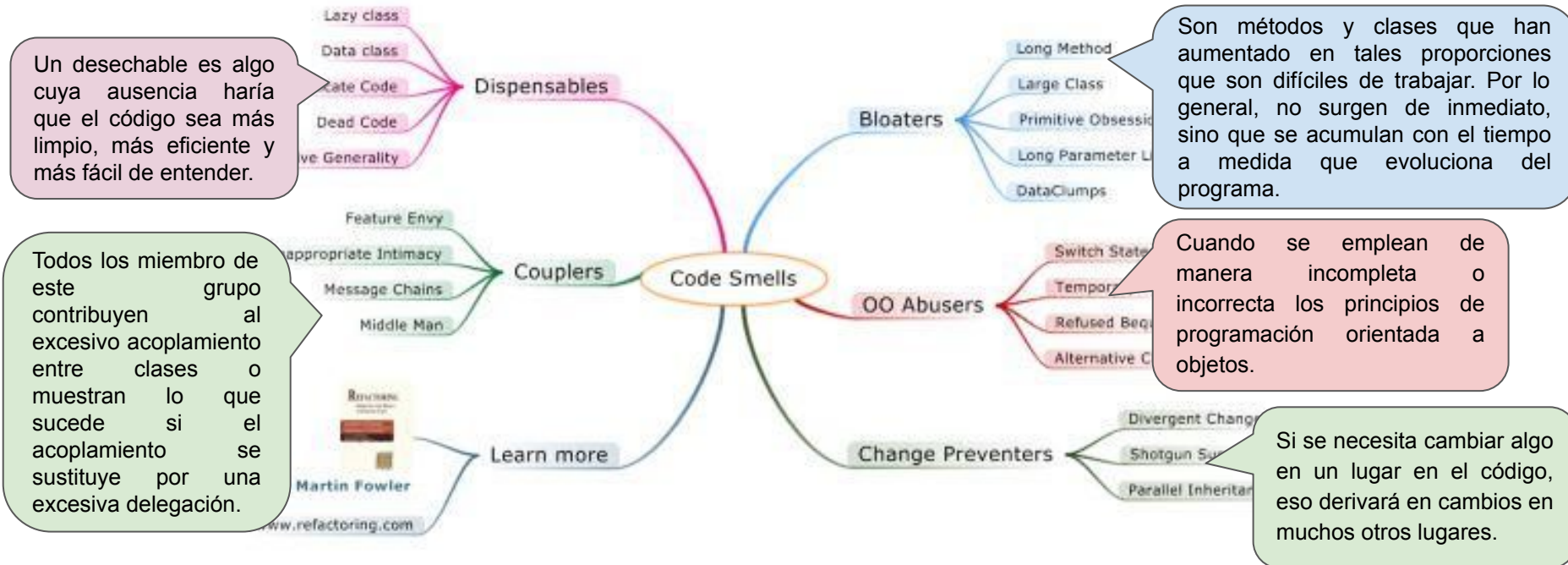
¿Cuándo hacer refactorización? Cuando hay código fuente que “huele mal” ...



Patrones de Diseño

Refactorización

¿Cuándo hacer refactorización? Cuando hay código fuente que “huele mal” ...



Desechables

Clase perezosa: La comprensión de las clases y su mantenimiento siempre cuesta tiempo y dinero. Así que si una clase no hace lo suficiente para ganar su atención, debe suprimirse.

Código duplicado: Dos fragmentos de código que son casi idénticos.

Código muerto: Una variable, parámetro, campo, método o clase ya no se utiliza (por lo general debido a que es obsoleto).

Clases de datos: Una clase de datos se refiere a una clase que contiene sólo campos primitivos y métodos para acceder a ellos (getters y setters). Estos son simplemente contenedores para datos. Estas clases no contienen ninguna funcionalidad adicional y no pueden operar de forma independiente.

Generalidad especulativa: Una clase, método, campo o un parámetro que no se utiliza.



Acopladores

Envidia de características:

Un método que accede a datos de otro objeto más que a los propios.

Intimidad inapropiada:

Una clase que utiliza campos internos o métodos de otra clase.

Cadena de mensajes:

Ocurre cadenas de mensajes cuando un cliente solicita un objeto por otro objeto, que a continuación, el cliente pide otro objeto, que el cliente pide entonces para otro otro objeto, y así sucesivamente.

Intermediario:

Si una clase realiza una sola acción y ésta delega el trabajo a otra clase, ¿Tiene sentido que exista?



Patrones de Diseño

Inflados

Método: Un método que contiene demasiadas líneas de código. Por lo general, un método no debería superar las 10 líneas de código.

Clase: Una clase que contiene demasiados campos, métodos, líneas de código.

Obsesión primitiva:

- Uso de primitivos en lugar de pequeños objetos para tareas simples (tales como dinero, rangos, cadenas especiales para números de teléfono, etc.)
- Uso de constantes para codificar información (por ejemplo la constante `USER_ADMIN_ROLE = 1`)

Listado de parámetros: Más de tres o cuatro parámetros para un método.

Grupos de código: A veces, diferentes partes del código contienen grupos idénticos de variables (conexión a una base de datos). Estos deben ser convertidos en sus propias clases.



Patrones de Diseño

Refactorización

Abusadores de la OO

Switchs:

Cuando hay un operador switch complejo o una secuencia de sentencias condicionales (if).

Campos temporarios:

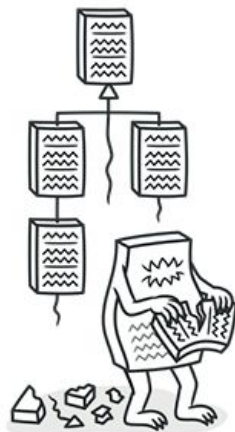
Campos temporarios obtienen sus valores sólo bajo ciertas circunstancias, fuera de ellas, están vacíos.

Legado rechazado:

(legado rechazado) Si una subclase emplea sólo algunos de los métodos o propiedades heredadas de sus padres, la jerarquía está desbalanceada.

Clases con diferentes interfaces:

Dos clases ejecutan idénticas funciones pero poseen métodos con diferentes nombres.



Obstaculizadores de cambios

Cambio divergente:

Cuando un cambio en una clase implica tener que hacer cambios en muchos métodos no relacionados. Por ejemplo, al agregar un nuevo tipo de producto hay que cambiar los métodos para encontrar, visualizar y ordenar productos.

Cirugía de escopeta:

Hacer cualquier modificación requiere de muchos pequeños cambios en muchas clases diferentes.

Jerarquías de herencia paralelas:

Siempre que se crea una subclase de una clase, se encuentra en la necesidad de crear una subclase de otra clase.



Patrones de Diseño

Refactorización



Code smells, olores de código o hediondez de código es una indicación superficial de algo que puede ser un problema más profundo en el sistema. El término fue acuñado por Kent Beck, mientras trabajaba junto con Martin Fowler para su libro Refactoring: Improving the Design of Existing Code.

En primer lugar un olor es por definición algo fácil de detectar. Un método extenso (cantidad de líneas de código) es un buen ejemplo de esto. Los olores de código no siempre indican un problema. Algunos métodos largos están bien así. Hay que mirar más profundo para ver si existe un problema subyacente. A menudo, los code smells, son un indicador de un problema más.

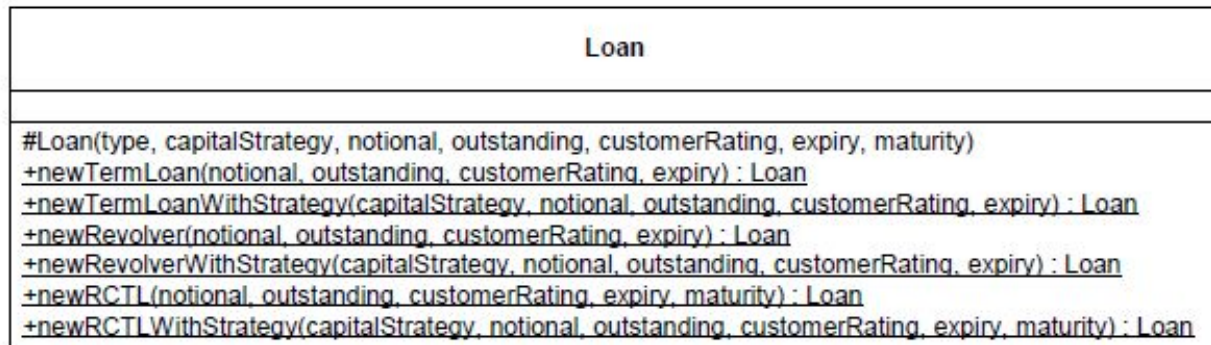
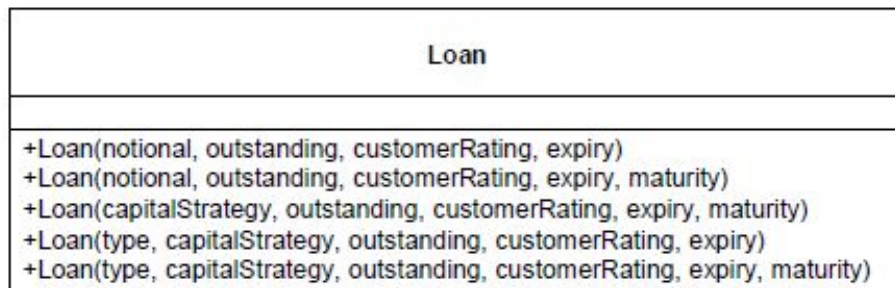
[Ejemplos en Java:](#)

<https://github.com/alejandroleoz/CodeSmellExamples>

Patrones de Diseño

Refactorización con Patrones

Caso 1: Reemplazar múltiples constructores con métodos creacionales



Patrones de Diseño

Refactorización

Caso 2:

Reemplazar un árbol
“ad hoc”
por un Composite

```
String orders = "<orders>";  
orders += "<order number='123'>";  
orders += "<item number='x1786'>";  
orders += "carDoor";  
orders += "</item>";  
orders += "</order>";  
orders += "</orders>";
```



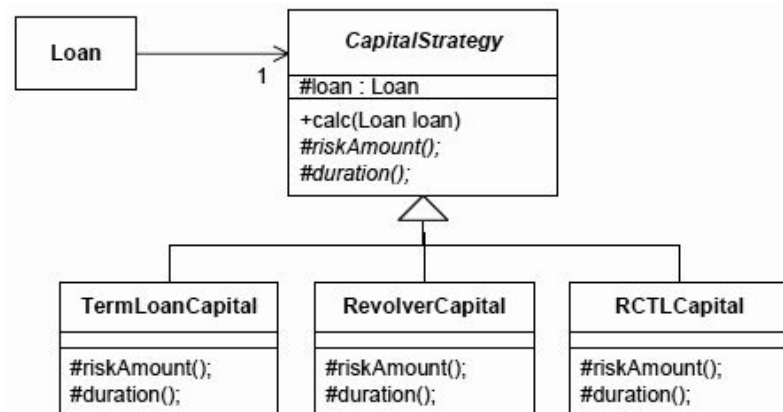
```
TagNode orders = new TagNode("orders");  
TagNode order = new TagNode("order");  
order.addAttribute("number", "123");  
orders.add(order);  
TagNode item = new TagNode("item");  
item.addAttribute("number", "x1786");  
item.addValue("carDoor");  
order.add(item);  
String xml = orders.toString();
```

Patrones de Diseño

Refactorización

Caso 3: Reemplazar cálculos condicionales por un Strategy

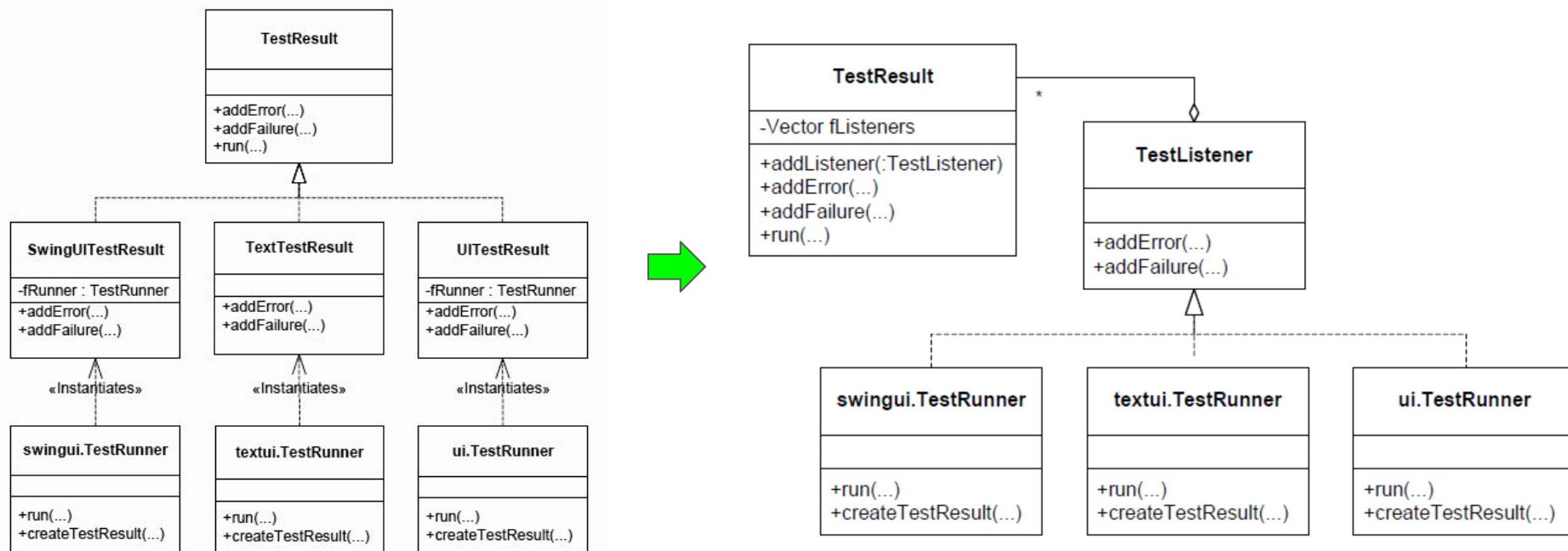
```
public class Loan ...
public double calcCapital() {
    return riskAmount() * duration() * RiskFactor.forRiskRating(rating);
}
private double riskAmount() {
    if (unusedPercentage != 1.00)
        return outstanding + calcUnusedRiskAmount();
    else return outstanding;
}
private double calcUnusedRiskAmount() {
    return (notional - outstanding) * unusedPercentage;
}
private double duration() {
    if (expiry == null)
        return ((maturity.getTime() - start.getTime())/MILLIS_PER_DAY)/365;
    else if (maturity == null)
        return ((expiry.getTime() - start.getTime())/MILLIS_PER_DAY)/365;
    else {
        long millisToExpiry = expiry.getTime() - start.getTime();
        long millisFromExpiryToMaturity = maturity.getTime() - expiry.getTime();
        double revolverDuration = (millisToExpiry/MILLIS_PER_DAY)/365;
        double termDuration = (millisFromExpiryToMaturity/MILLIS_PER_DAY)/365;
```



Patrones de Diseño

Refactorización

Caso 4: Cambiar notificaciones “hard-codeadas” por un Observer

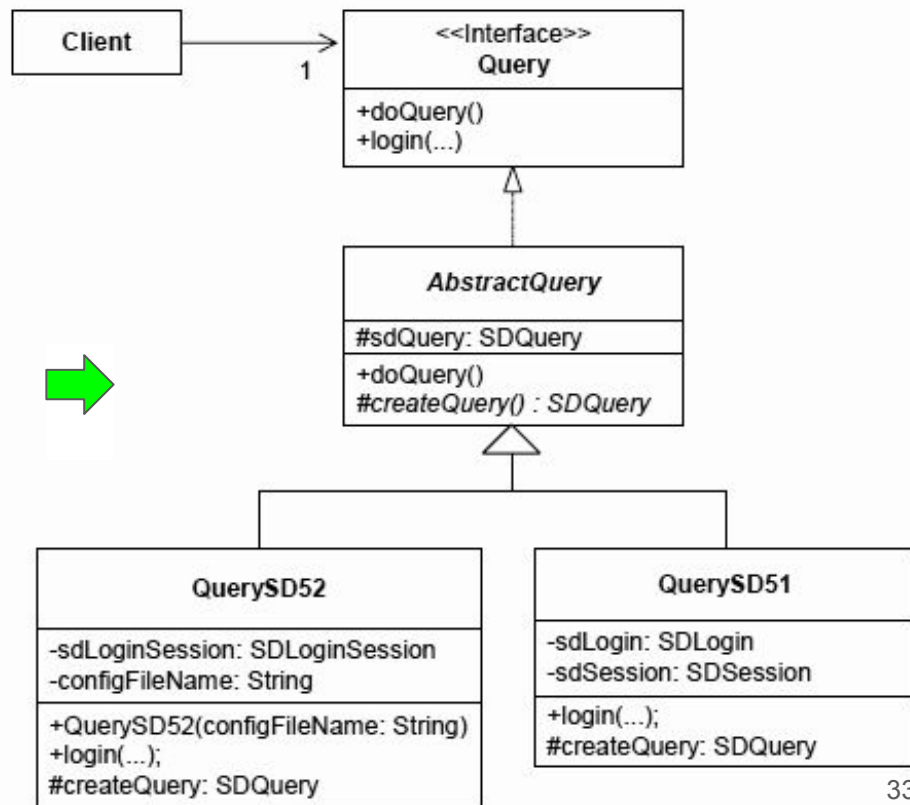
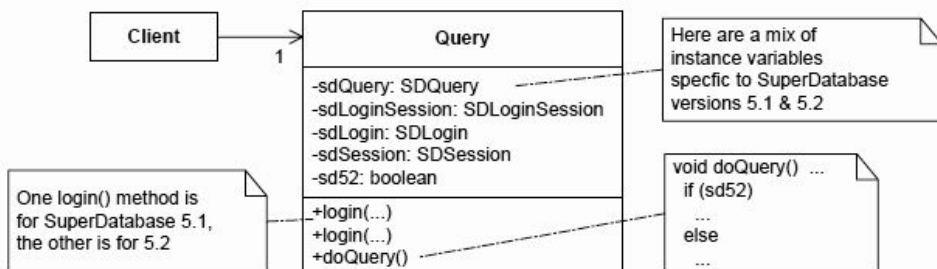


Patrones de Diseño

Refactorización

Caso 5:

Separar versiones con un Adapter



Referencias

Balaguer, F., Garrido, A., Introducción a Patrones de Diseño.

Tópicos de Ingeniería de Software II. Postgrado Universidad Nacional de La Plata. 2011.

Gamma, E. et al., Design Patterns: Elements of Reusable Object-Oriented Software.

Addison-Wesley. 1994.

Metsker, S., The Design Patterns Java Workbook.

Addison-Wesley. 2002.

Fowler, M., Refactoring, Improving the Design of Existing Code.

Addison-Wesley. 1999.

Kerievsky, J., Refactoring to Patterns.

Addison-Wesley. 2004.

Source Making.

<https://sourcemaking.com/refactoring>