Refactorización

IXX

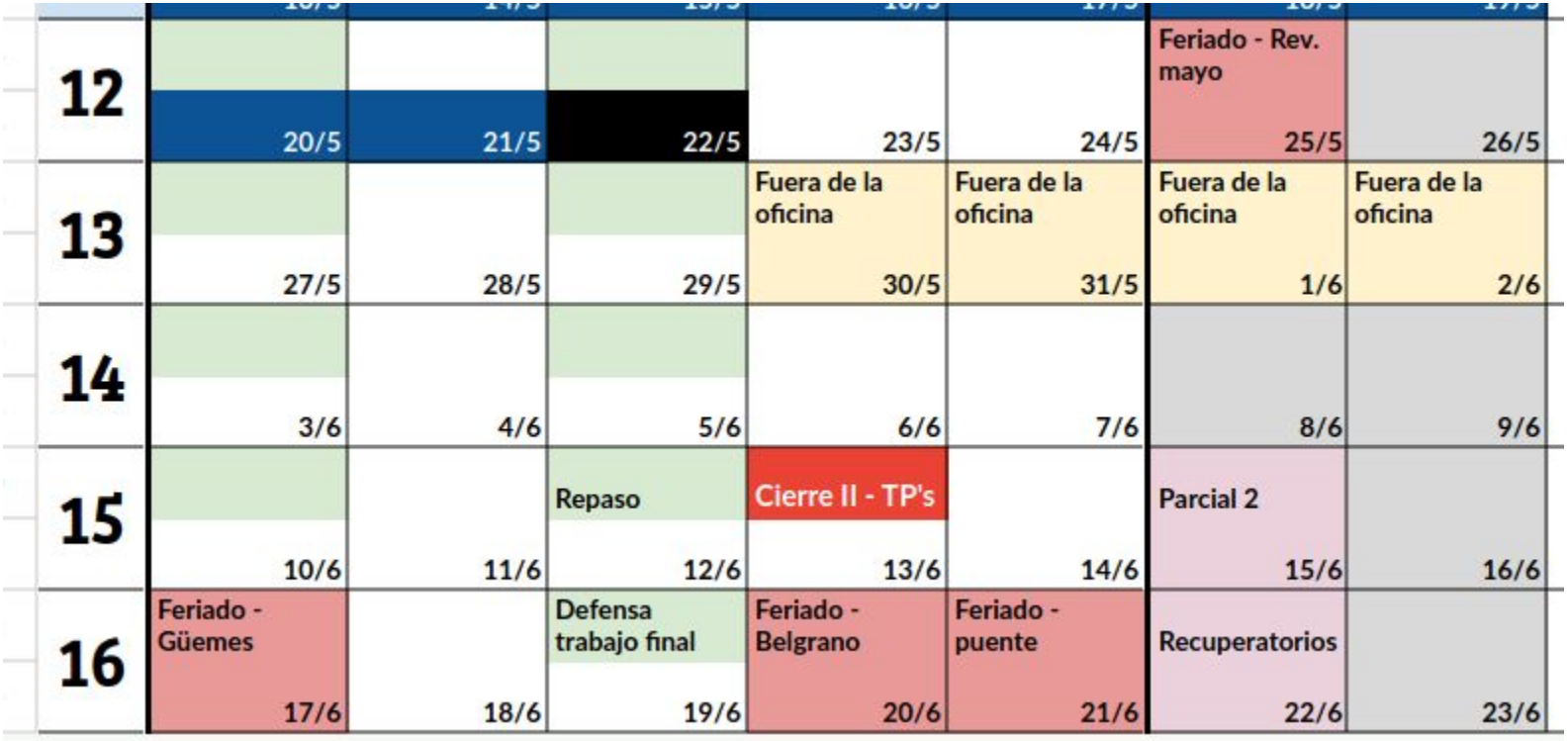
2024

UNRN

Universidad Nacional
de **Río Negro**

r17

Parcial II



12	20/5	21/5	22/5	23/5	24/5	Feriado - Rev. mayo 25/5	26/5
13	27/5	28/5	29/5	Fuera de la oficina 30/5	Fuera de la oficina 31/5	Fuera de la oficina 1/6	Fuera de la oficina 2/6
14	3/6	4/6	5/6	6/6	7/6	8/6	9/6
15	10/6	11/6	Repaso 12/6	Cierre II - TP's 13/6	14/6	Parcial 2 15/6	16/6
16	Feriado - Güemes 17/6	18/6	Defensa trabajo final 19/6	Feriado - Belgrano 20/6	Feriado - puente 21/6	Recuperatorios 22/6	23/6

Se termina el cuatrimestre

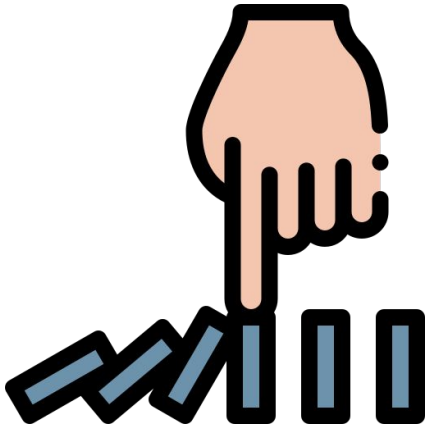
Queda por ver Introducción a Estructuras de datos


¿Preguntas?

Acoplamiento

Acoplamiento fuerte

Es la dependencia en los detalles internos de otras clases.



La ley de Demetrio

(no hables con extraños)

¿Quiénes son amigos?

El propio objeto

Argumentos en métodos

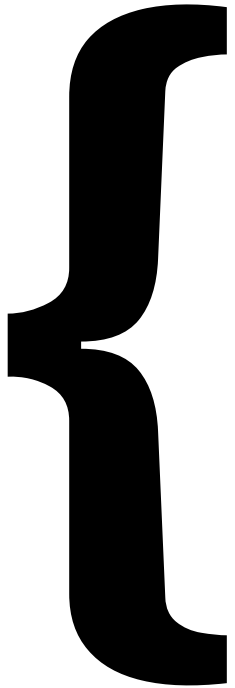
Objetos creados en un método

Objetos empleados como atributos

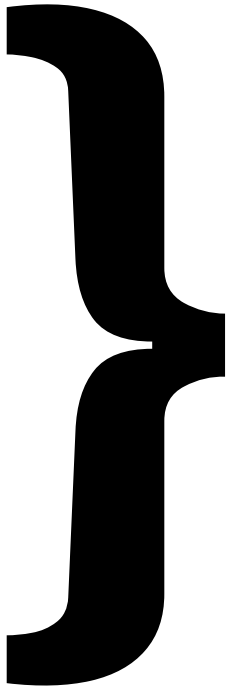
¿Quiénes son extraños?

Objetos obtenidos en la llamada de otros objetos

Objetos obtenidos al acceder atributos de otros objetos



los getters y setters



Dado un Contacto (con dirección)

```
item.getDireccion().getCiudad().getCalle().getAltura();
```

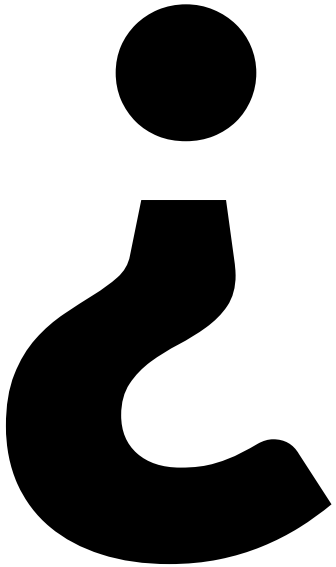
Nos acoplan a la estructura

dirección/ciudad/calle/altura

Y no se resuelve usando variables!

¡Esto es lo mismo que lo anterior!

```
Direccion d = item.getDireccion()
Ciudad c.getCiudad();
Calle s = c.getCalle()
Altura a = s.getAltura();
```



Como lo resolvemos

Dile que hacer, no preguntes

Contra-ejemplo

```
public class Empleado {
    private Departamento departamento;
    // ...
    public void enviarCorreoInformativo() {
        if (departamento.getJefe().estaDisponible()) {
            departamento.getJefe().enviarCorreo(this,
                    "Información importante");
        }
    }
}
```

Una potencial solución

```
public class Empleado {
    private Departamento departamento;
    // ...
    public void enviarCorreoInformativo() {
        departamento.enviarCorreo(this,
            "Información importante");
        }
    }
}
```

Lo que es la búsqueda en Agenda/Contacto


¿Preguntas?

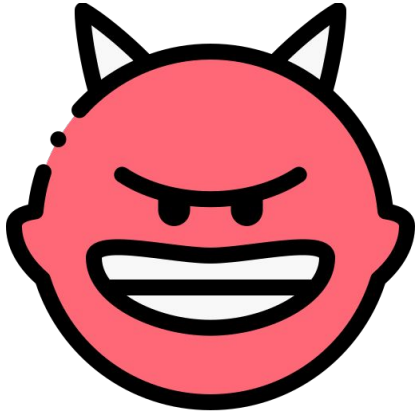
Otro principio importante

Principio Hollywood

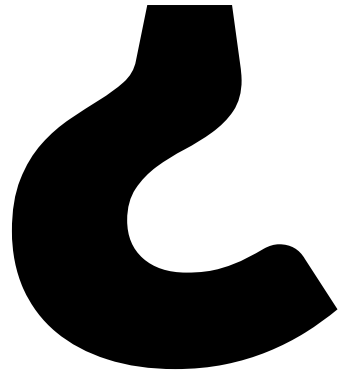
Inversión del Control

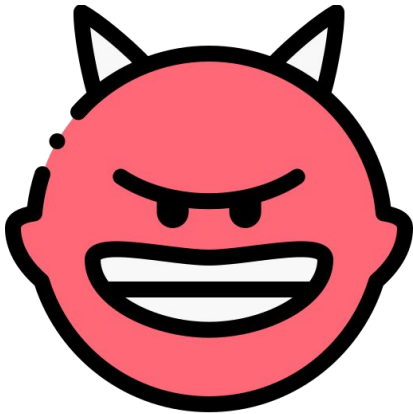
Como Observer

```
public class Aplicacion {
    private Servicio servicio;

    public Aplicacion(Servicio servicio) {
        this.servicio = servicio;
    }

    public void realizarTarea() {
        servicio.ejecutar();
    }
}
```



Esto quiere decir que los getters/setters son malignos





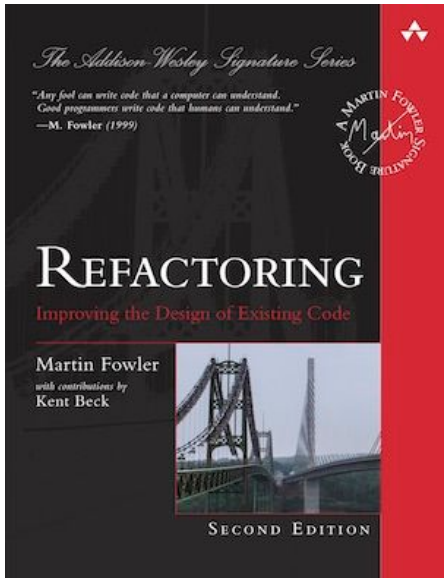

nope

Pero tenemos que evitar su uso tanto como sea posible


¿Preguntas?

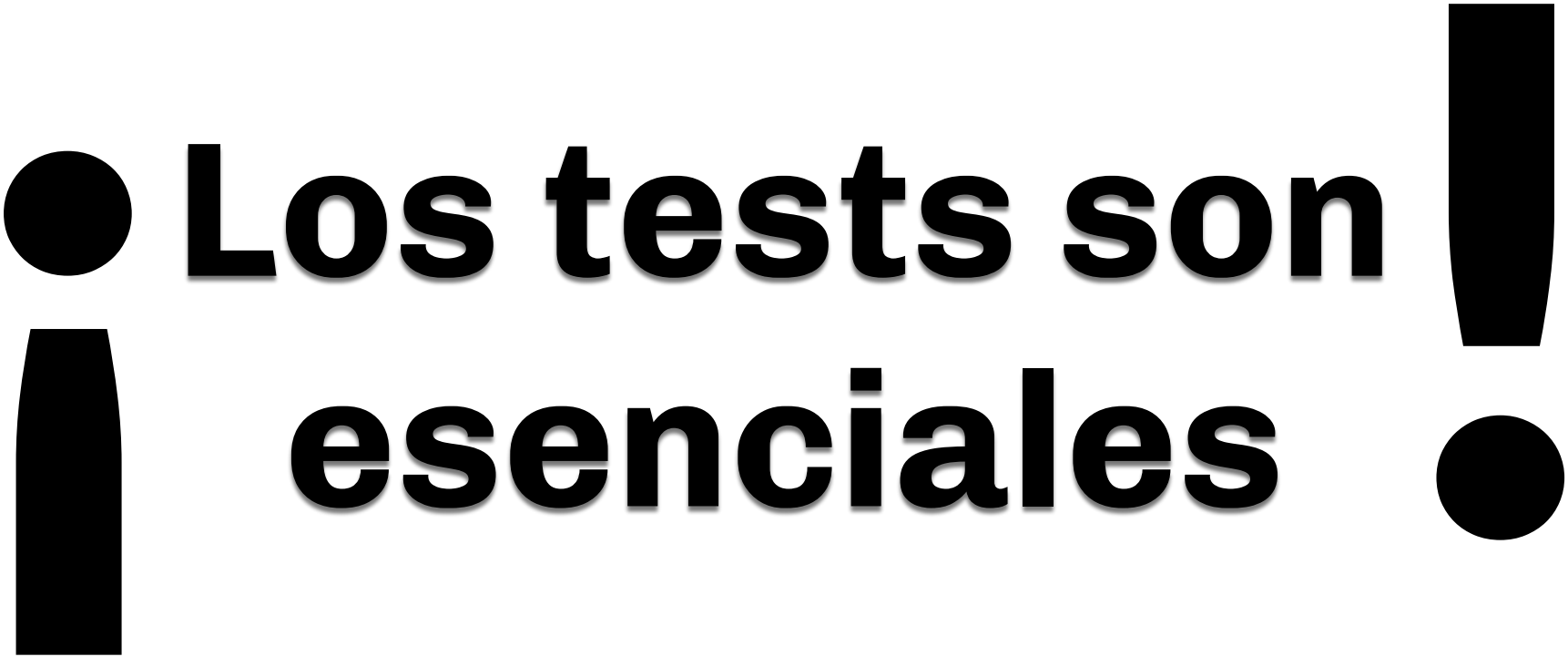
el plan de pagos

Refactorización



Refactorizar es

el proceso de reestructurar el código existente sin cambiar su comportamiento externo.


¡Los tests son esenciales!

{ Ayudan a garantizar que los cambios no alteran el funcionamiento }

Refactorizaciones por ámbito

De clase

Extraer clase

C

1

Divide una clase grande y compleja en varias clases más pequeñas y cohesivas, cada una con una responsabilidad única.

¿Cuándo aplicar?

- Clase sobrecargada
- Grupo de datos cohesionado
- Reutilización de código

Como aplicar

Identificar el grupo de datos

Crear la nueva clase

Establecer la nueva relación

Ajustar las referencias

Probar exhaustivamente

En Contacto (TP9)

```
public class Contacto{
    private int diaNacimiento;
    private int mesNacimiento;
    private int añoNacimiento;

// metodos relacionados a la manipulación de esos atributos.
…
```

Es algo como una FechaNacimiento

```
public class FechaNacimiento {
    private int diaNacimiento;
    private int mesNacimiento;
    private int añoNacimiento;
… Con todos los métodos que le correspondan.
```

Otro ejemplo

```
class Persona {
    String nombre;
    int edad;
    String calle;
    int numero;
    String ciudad;
    String codigoPostal;

    void cambiarDireccion(String nuevaCalle, int nuevoNumero, String
nuevaCiudad, String nuevoCodigoPostal) {
        // ...
    }
}
```

Y así podemos seguir extrayendo.

```
class Direccion {
    String calle;
    int numero;
    String ciudad;
    String codigoPostal;

    void cambiarDireccion(String nuevaCalle,
                          int nuevoNumero,
                          String nuevaCiudad,
                          String nuevoCodigoPostal) {
        // ...
    }
}
```

Cuando no usarlo

- Poca cohesión
- Dependencias complejas
- Clase chiquita


¿Preguntas?

Mover método

C

2

Mueve un método de una clase a otra donde tenga más sentido según su funcionalidad.

¿Cuándo aplicar?

- Método fuera de lugar
- Clase repleta de métodos
- Reutilización de código

Como aplicar

1. Identificar el método candidato
2. Elegir la clase destino
3. Crear el método en la clase destino
4. Reemplazar la llamada original
5. Probar exhaustivamente

Cuando no usarlo

- Dependencias complejas
- Método privado
- Método estático


¿Preguntas?

Extraer superclase

C

3

Crea una nueva clase base para agrupar funcionalidades comunes de varias clases existentes.

¿Cuándo aplicar?

Duplicación de código: Cuando dos o más clases comparten atributos o métodos idénticos o muy similares.

Como aplicar

1. Identificar elementos comunes
2. Crear la superclase
3. Establecer la herencia
4. Reemplazar duplicados
5. Ajustar constructores
6. Probar exhaustivamente

Ejemplos

```
class CajaDeAhorro {
    …
    void depositar(Dinero plata){…}
}

class CuentaCorriente {
    …
    void depositar(Dinero plata){…}
}
```

Si la operación es la misma, entonces debe ser la misma

```
class CajaDeAhorro
    extends Cuenta{
     …
}

class CuentaCorriente
     extends Cuenta{
     …

}
```

```
class Cuenta{
     …
     void depositar(Dinero plata){…}
}
```

Más orientado a los atributos

```
class Perro {
    String nombre;
    int edad;
    String raza;

    void comer() {…}
    void dormir() {…}
}
```

```
class Gato {
    String nombre;
    int edad;
    String raza;

    void comer() {…}
    void dormir() {…}
}
```

Perro y gato pueden sobreescribir `comer` y `dormir`

```
class Animal {
    String nombre;
    int edad;
    String raza;

    void comer() { ... }
    void dormir() { ... }
}
class Perro extends Animal { }
class Gato extends Animal { }
```

Cuando no usarlo

- Poca similitud
- Jerarquía forzada
- Acoplamiento excesivo


¿Preguntas?

Reemplazar condicional con polimorfismo

C

4

Utiliza el polimorfismo en lugar de condicionales para hacer el código más flexible y extensible.

(ver Calculadora)

¿Cuándo aplicar?

- Condicionales basados en tipo
- Código duplicado
- Jerarquía de tipos

Como aplicar

1. Identificar el condicional
2. Crear la jerarquía de clases
3. Crear las subclases
4. Reemplazar el condicional
5. Probar exhaustivamente

Ejemplo

```
void procesarPago(String tipoPago, double monto) {
    if (tipoPago.equals("tarjeta")) {
        // Lógica para procesar pago con tarjeta de crédito
    } else if (tipoPago.equals("paypal")) {
        // Lógica para procesar pago con PayPal
    } else if (tipoPago.equals("transferencia")) {
        // Lógica para procesar pago con transferencia bancaria
    } else {
        throw new IllegalArgumentException("Tipo de pago no
válido");
    }
}
```

Refactorizado en

```
interface ProcesadorPago {
    void procesar(double monto);
}

class ProcesadorTarjeta implements ProcesadorPago {
    public void procesar(double monto) {
        // Lógica para procesar pago con tarjeta de crédito
    }
}
```

Mas todos los que sean necesarios.

Cuando no usarlo

- Pocos tipos (o una cantidad acotada)
- Condicional simple
- Tipos no relacionados


¿Preguntas?

Introducir Interfaz

C

5

Define una interfaz para desacoplar clases y mejorar la flexibilidad.

¿Cuándo aplicar?

Comportamiento común, pero la herencia no es clara

Desacoplamiento

Polimorfismo

Tests

Como aplicar

1. Identificar el conjunto de métodos
2. Crear la interfaz
3. Implementar la interfaz
4. Utilizar la interfaz
5. Probar exhaustivamente

Cuando no usarlo

Pocos métodos comunes

Escasa relación conceptual

Interfaz redundante


¿Preguntas?

Reemplazar herencia por composición

C

6

Usa composición en lugar de la herencia para lograr una mayor flexibilidad y evitar problemas de acoplamiento.

¿Cuándo aplicar?

Subclase poco especializada

Acoplamiento excesivo

Jerarquia de clases rígida

Como aplicar

1. Crear un atributo delegador
2. Delegar los métodos
3. Eliminar la herencia
4. Ajustar los constructores
5. Probar exhaustivamente

Ejemplo

```
class Empleado extends Persona {
// ... atributos y métodos específicos de Empleado
}
```

Refactorizado

```
class Empleado {
    private Persona persona;

    public Empleado(Persona persona) {
        this.persona = persona;
    }
    // ... atributos y métodos específicos de Empleado
}
```

Cuando no usarlo

Jerarquia de clases adecuada

Polimorfismo escencial


¿Preguntas?

de método

Extraer método

M

1

Divide un método largo y complejo en varios métodos más pequeños y enfocados.

En arreglos, la acción de copiar arreglo

Introducir parámetro

M

2

Añade un nuevo parámetro a un método para hacerlo más flexible y reutilizable. Este puede ser una sobrecarga.

Eliminar parámetro

M

3

Elimina un parámetro de un método si ya no es necesario. Puede ser tambien como sobrecarga.

Reemplazar parámetro con método

M

4

Reemplaza un parámetro por una llamada a un método para mejorar la legibilidad y reducir la duplicación de código.

Reemplazar parámetro con atributo

M

5

Introduce un atributo para que este ya forme parte de la clase y sea utilizado por los métodos.

Dividir

M

6

Tomar un fragmento de código dentro de una función o método existente y convertirlo en una nueva función o método independiente.

de variable

Introducir variable explicativa

V1

Crea una nueva variable para almacenar el resultado de una expresión compleja y mejorar la legibilidad del código.

Dividir variable temporal

V2

Divide una variable temporal en varias variables si se utiliza para almacenar diferentes valores a lo largo del tiempo.

Eliminar asignaciones a parámetros

V

3

Evita modificar los valores de los parámetros de un método, ya que esto puede generar confusión.

generales

Renombrar

clase/método/variable/constante

G

1

Cambia el nombre de una entidad para que refleje mejor su propósito o siga el estándar.

¿Cuándo aplica?

Identificador poco claro.

Abreviaturas

Cambio de propósito

Convenciones

¿Cuándo no aplica?

Nombre ampliamente utilizado (`i`)

Nombre claro y preciso

Clases internas o anónimas

Dividir condicional

G

2

Simplificar una estructura condicional compleja (como múltiples if-else anidados) dividiéndola en funciones o métodos más pequeños y legibles.

¿Cuándo aplica?

Condición anidada

Condición larga

Condición con múltiples responsabilidades

Usen métodos para darles nombre

```
if (nota >= 4)
```

```
if estaAprobado(nota)
```


¿Preguntas?

¿Qué hacer?

¿Cuándo aplicar?

Como aplicar

Cuando no usarlo


¿Preguntas?

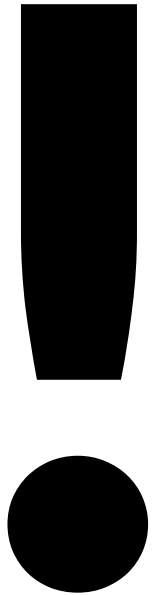
{ Estas son una fracción }

refactoring.com/catalog/

Hide Delegate
Inline Class
Inline Function
Inline Variable
Introduce Assertion
Introduce Parameter Object
Introduce Special Case
Move Field
Move Function
Move Statements into Function
Move Statements to Callers
Parameterize Function
Preserve Whole Object
Pull Up Constructor Body
Pull Up Field
Pull Up Method
Push Down Field
Push Down Method
Remove Dead Code
Remove Flag Argument
Remove Middle Man
Remove Setting Method
Remove Subclass
Rename Field
Rename Variable
Replace Command with Function
Replace Conditional with Polymorphism
Replace Constructor with Factory
Replace Control Flag with Break
Replace Derived Variable with Query
Replace Error Code with Exception
Replace Exception with Precheck
Replace Function with Command
Replace Inline Code with Function Call
Replace Loop with Pipeline
Replace Magic Literal
Replace Nested Conditional with Guard Clauses
Replace Parameter with Query
Replace Primitive with Object
Replace Query with Parameter
Replace Subclass with Delegate
Replace Superclass with Delegate
Replace Temp with Query
Replace Type Code with Subclasses
Return Modified Value
Separate Query from Modifier
Slide Statements
Split Loop
Split Phase
Split Variable
Substitute Algorithm

Algunas refactorizaciones son la aplicación de patrones

Igual, no las vamos a ver




¿Preguntas?

Pero al igual que los patrones

Desarrollan terminología **muy** específica


¿Preguntas?

Refactorizaciones

unrn.edu.ar

UNRN Universidad Nacional de Río Negro

| unrionegro