

Refactoring

LEYES DE LEHMAN

las leyes de Lehman son un conjunto de leyes empíricas relacionadas con la evolución del software pero que pueden ser aplicadas a cualquier cosa software o hardware

- continuing change: los sistemas deben adaptarse continuamente o se vuelven menos satisfactorios
- continuing growth: la funcionalidad debe ser incrementada continuamente para mantener satisfacción
- increasing complexity: mientras que el sistema evoluciona, su complejidad aumentará
- declining quality: la calidad de un sistema va a ir declinando a menos que se haga un mantenimiento riguroso

a veces se pierden oportunidades de negocio porque el ritmo de cambio es exponencial, lo que requiere tiempo de reacción más rápido

MANTENIMIENTO

el mantenimiento de software es la modificación de un producto de software después de la entrega, para corregir errores, mejorar el rendimiento, u otros atributos

el mantenimiento resulta ser:

- correctivo: mantenimiento de software correctivo es necesario cuando algo sale mal en una pieza de software, incluidos fallos y errores
- evolutivo: proceso realizado para aumentar, disminuir o cambiar las funcionalidades del sistema
- adaptativo: tiene que ver con las tecnologías cambiantes, así como con las políticas y reglas relacionadas con su software. Las cuales incluyen cambios en el sistema operativo, almacenamiento en la nube, hardware, etc.
- perfectivo: tiene como objetivo ajustar el software agregando nuevas características según sea necesario y eliminando características que son irrelevantes o no efectivas en el software dado
- preventivo: el mantenimiento preventivo de software está mirando hacia el futuro para que su software pueda seguir funcionando como se deseó durante el mayor tiempo posible. Esto incluye realizar los cambios necesarios, actualizaciones, adaptaciones y más

BIG BALL OF MUD

big ball of mud es el código mal hecho o estructurado, con crecimiento descontrolado y que se mantiene a base de alambres y existen porque funcionan pasando los tests

THROWAWAY CODE

el throwaway code es el código desecharable para hacer un prototipo o mostrar un concepto. Es simple, predictible y descartable

PIECEMEAL GROWTH

el piecemeal growth es el proceso de diseño e implementación en el que el software se embellece, modifica, reduce y mejora mediante el proceso de mejora y *no* de modificación

en conclusión, diseñar resulta difícil porque los elementos distintivos de la arquitectura no surgen hasta después de tener un código que funciona, no solo es agregar sino también adaptar, transformar y mejorar y los errores y cambios resultan inevitables por lo tanto hay que aprender del *feedback* el código reusable es el resultado de iteraciones de diseño

METODOLOGÍAS ÁGILES

las metodologías ágiles envuelven un enfoque para la toma de decisiones en los proyectos de software, refiriéndose a métodos basados en el desarrollo iterativo e incremental

CARÁCTERÍSTICAS

- mismo grupo de personas para todo el desarrollo
- comunicación de calidad
- producto funcionando en cada "build"
- es importante el feedback
- los cambios son bienvenidos

TEST DRIVEN DEVELOPMENT

el test driven development es una práctica que consiste en escribir primero las pruebas(*test first development*), después escribir el código fuente que pase la prueba satisfactoriamente, y por último *refactorizar* el código escrito

su objetivo es pensar en el diseño y qué se espera de cada requerimiento antes de codear

- test de aceptación: por cada funcionalidad esperada, escritos desde la *perspectiva del cliente*. Usados para capturar los casos de uso
- test de unidad: aísla cada unidad de un programa y muestra que funciona correctamente, escritos desde la *perspectiva del programador*. Se enfoca en pequeñas partes a la vez y aísla los errores.

no debo dejar el testing para el final, es mejor hacerlo antes porque así puedo refactorizar rápido, tengo más confianza de que voy por buen camino y es una medida de progreso

- el TDD es un cambio al desarrollo de software tradicional donde primero escribo los test en vez de escribir el código
- no se agrega funcionalidad hasta que no haya un test
- una vez escrito el test, se codifica lo necesario para que todo el test pase
- pasados los test, se refactoriza para asegurar calidad del código

REGLAS

- diseñar incrementalmente, teniendo un código que funciona como feedback
- los programadores escriben sus propios tests
- el diseño debe consistir de componentes cohesivos y desacoplados

LOGROS

diseño simple y limpio, desarrollo más rápido, confianza, documentación práctica, más calidad de software (el software está construido correctamente y es correcto)

TEST DE UNIDAD

un test de unidad realiza un testeo de la mínima unidad de ejecución

en POO la mínima unidad es un método. Tiene como objetivo aislar cada parte de un programa y mostrar que funciona correctamente
un testing asume la presencia de un framework que automatiza tests (ejecutar muchas veces los test creados y ver su resultado fácilmente)
un test de unidad está formado por:

1. fixture set up: prepara todo lo necesario para testear el comportamiento del SUT. Incluye el código para instanciar el SUT, ponerlo en estado apropiado y el código para crear e inicializar todo aquello de lo que el SUT depende o es pasado como argumento
2. exercise: interactúo con el SUT para ejercitarse el comportamiento que se intenta verificar
3. check: comprobar si los resultados son los esperados
4. tear down: limpiar los objetos creados para y durante el test

patrones del fixture setup

- in-line setup: cada test crea su propio fixture nuevo
- delegated setup: cada test crea su fixture llamando a métodos de creación
- implicit setup: se construye un fixture común a varios tests en el método `setUp()`, como vengo haciendo hasta ahora

al testear funcionalidades del SUT (system under test) es preferible no depender de componentes del sistema ajenos al test, por lo tanto hacemos **test doubles** o **mock objects** que son simuladores que imitan el comportamiento de otros objetos de manera controlada

- dummy object: usa el objeto para que ocupe un lugar pero nunca es usado
- test stub: sirve para que el SUT le envíe los mensajes separados y devuelva un valor por defecto
- test spy: test stub + registro de outputs
- mock object: test stub + verification of outputs
- fake objects: imitación, se comporta como el módulo real

cuando el objeto real es complejo (retorna resultados no determinísticos, tiene estados difíciles de reproducir, es lento, todavía no existe, tiene dependencias con otros objetos y necesita ser aislado para testearlo como unidad)

REGLAS DEL TESTING

- mantener los tests independientes entre sí
- un buen test es simple, fácil de escribir y mantener
- su objetivo es encontrar bugs
- tiene ciertas limitaciones, no encuentra todos los errores, no puede comprobar la ausencia de errores

- [“xUnit Test Patterns: Refactoring Test Code”. Gerard Meszaros.](#)
- “Test Driven Development by Example”. by Kent Beck.
- [“The Little Mocker”. Robert Martin.](#)
- [Video: Google+ talk con Kent Beck y Martin Fowler sobre los problemas del TDD](#)
- “Big Ball of Mud”. Brian Foote and Joe Yoder. Pattern Languages of Programs 4. Addison-Wesley 2000.
- [Big ball of mud @ Google Talks 2007](#)
- “Refactoring”. Martin Fowler. Addison-Wesley 1999. (original con ejemplos en Java)
- “Refactoring. 2nd edition”. Martin Fowler. Addison-Wesley 2018. (ejemplos en JavaScript)
- [Martin Fowler @ OOP2014 "Workflows of Refactoring"](#)

REFACTORING

el refactoring es una transformación que preserva el comportamiento pero mejora el diseño

se caracteriza por mejorar la organización, legibilidad, adaptabilidad y adaptabilidad del código luego que fue escrito, lo importante es que NO altera el comportamiento. Implica eliminar duplicaciones, simplificar lógica y clarificar código.

El refactoring es aplicado cuando el código funciona y pasa los test o mientras estoy desarrollando y encuentro código difícil o tengo que hacer cambios y reorganizar.

ayuda introduciendo mecanismos que solucionan problemas de diseño mediante pequeños cambios porque es más fácil que hacer uno grande.
es importante porque es una defensa contra el deterioro del software, facilita la incorporación de código.

surge una **deuda técnica**, el cual refleja el costo implícito del retrabajo adicional causado por elegir una solución fácil en lugar de utilizar un enfoque que llevaría más tiempo en su desarrollo e implementación.

el refactoring es importante porque gano comprensión de código, reduzco costo de mantenimiento debido a los cambios inevitables que sufrirá el sistema, facilito la detección de bugs

CLEAN CODE: cohesive, loosely coupled, encapsulated, assertive, non-redundant

se toma que un código **huele mal** producto de mal diseño como código duplicado, ilegible o complicado, y se trabaja para tener un mejor diseño, moviendo atributos o métodos de una clase a otra, extrayendo código de un método en otro método, moviendo código en la jerarquía, etc.

el refactoring se aplica en el contexto de TDD, al descubrir un código con mal olor, cuando no puedo entender el código o cuando encuentro una mejor forma de codificar algo

THE 2 HATS: *adding function* (exploro ideas y corrojo bugs) o *refactoring* (solo refactorizo con test en verde)

“Refactoring. Improving the Design of Existing Code”. Martin Fowler. Addison Wesley. 1999.

Sitio de refactoring: refactoring.com

“Object Oriented Metrics in Practice”. Lanza & Marinescu. Springer 2006

BAD SMELLS

los bad smell son indicios de problemas que requieren aplicación de refactorings

- 1) bloaters: es código, métodos y clases que crecen gigantescamente y es complicado trabajar con ellos. Estos bad smell surgen con el paso del tiempo a medida que el programa evoluciona
- 2) change preventers: si necesito cambiar algo en un lugar de mi código, también deberé realizar cambios en otros lugares. El desarrollo del programa se vuelve más complicado y costoso
- 3) couplers: estos bad smells contribuyen al acoplamiento excesivo entre clases o muestran lo que sucede si el acoplamiento se reemplaza por una delegación excesiva
- 4) tool abusers: bad smells que implementa mal o de forma incompleta los principios de la POO
- 5) dispensables: es un bad smell de algo que no tiene sentido y su ausencia haría que el código fuera más limpio, eficiente y fácil de entender

bloaters	change preventers	couplers	tool abusers	dispensables
long method	divergent change	feature envy	switch statements	lazy class
large class	shotgun surgery	inappropriate intimacy	refused bequest	speculative generality
data clumps	parallel inheritance	message chains	alternative with classes	data class
long parameter list		middle man	temporary field	duplicate code
primitive obsession				

- **long method:** es un método que tiene muchas líneas de código. Se considera que es largo cuando tiene más de 20 o 30 líneas, depende del lenguaje de programación. Resulta difícil de entenderlo, cambiarlo y reusarlo
 - extract method, decompose conditional, replace temp with query
- **large class:** una clase intenta hacer mucho trabajo, tiene muchas variables de instancia o métodos. Generalmente es código duplicado
 - extract class, extract subclass
- **data clumps:** a veces, diferentes partes del código contienen grupos idénticos de variables (como parámetros para conectarse a una base de datos). Estos grupos deben convertirse en sus propias clases
 - extract class, introduce parameter object, preserve whole object
- **long parameter list:** un método con una larga lista de parámetros es más difícil de entender, también es difícil de obtener todos los parámetros para pasarlos en la llamada entonces el método es difícil de reusar
 - replace parameter with method, preserve whole object, introduce parameter object
- **primitive obsession:** usa primitivos en vez de pequeños objetos como listas, arrays, etc. Si tengo una gran cantidad de campos primitivos, sería lógico agruparlos de alguna forma
 - replace data value with object, introduce parameter object / preserve whole object (si el valor es usado en parámetros de métodos), replace array with object

- **divergent change:** si un cambio en una clase requiere cambiar varias partes de la clase
 - extract class
- **shotgun surgery:** si un cambio en una clase requiere cambiar en cascada en varias clases
 - move method/field
- **parallel inheritance:** ocurre cuando las subclases de la clase A deben estar vinculadas a las subclases de la clase B
 - move method, move field

- **feature envy:** un método de una clase usa los datos y métodos de otra clase para hacer su trabajo, este bad smell indica que el método fue ubicado en una clase incorrecta
 - move method
- **inappropriate intimacy:** una clase usa campos y métodos de otra clase
 - move method/field
- **message chains:** ocurre cuando un objeto llama a otro objeto, este llama a otro y sucesivamente
 - hide delegate, extract method y move method
- **middle man:** si tengo una clase que solo realiza una acción, delegando trabajo a otra clase debería sacarla
 - remove middle man

- **switch statements:** uso de operador switch o if que se vuelve complejo
 - replace conditional with polymorphism
- **refused bequest:** relacionado al uso de herencia y puede ser visto en clases derivadas que no usan la mayoría de las funcionalidades de la clase padre. Incluso teniendo esa herencia, la clase padre y la clase derivada no se parecen
 - push down method/field
- **alternative class:** dos clases realizan idénticas funciones pero con nombres de métodos diferentes
 - rename method (para que los nombres sean idénticos), move method, extract superclass

- **temporary field:** obtienen sus valores (y, por lo tanto, los objetos los necesitan) solo en determinadas circunstancias. Fuera de estas circunstancias, están vacíos
 - extract class, replace method with method object
- **lazy class:** es una clase que rara vez se usa y podría ser innecesaria y su funcionalidad podría formar parte de otra clase. Lazy class es el bad smell opuesto a Large class
 - inline class (componentes que casi ni se usan), collapse hierarchy (para subclases con nuevas funciones)
- **speculative generality:** hay clases, métodos o variables sin usar que estaría teniendo "por las dudas"
 - collapse hierarchy (para clases abstractas sin usar), inline class (para delegación de funcionalidades a otras clases), remove parameter (para métodos son parámetros que no uso)
- **data class:** una clase que solo tiene variables y getters/setters. Sucede cuando otras clases tienen métodos con "envida de atributo", indica que esos métodos deberían estar en la "data class"
 - move method
- **duplicate code:** mismo código o muy similar. Hará que el código más largo de lo que tiene que ser, es difícil de mantener y cambiar
 - extract method, pull up method, form template method
- **comentarios**
 - extract method, rename method

FEATURE ENVY VS INAPPROPRIATE INTIMACY

- feature envy ocurre cuando un método usa más características de otra clase que las propias y como consecuencia conduce que el diseño se enrede entre dos clases
- inappropriate intimacy compromete la encapsulación de la otra clase. Por ejemplo accediendo directamente a las variables de instancia a las que no se debe acceder directamente

DIVERGENT CHANGE VS SHOTGUN SURGERY

- en divergent change es cuando muchos cambios son hechos en una sola clase
- en shotgun surgery se refiere a cuando un solo cambio en una clase genera muchos cambios en otras clases

CATÁLOGO DE REFACTORINGS

permiten distribuir el código adecuadamente	mover aspectos entre objetos para mejorar las responsabilidades	mejoran la jerarquía de clases	organizan los datos	simplifican condicionales	simplifican la invocación de métodos
<ul style="list-style-type: none"> • extract method • inline method • replace temp with query • split temporary variable • replace method with method object • substitute algorithm 	<ul style="list-style-type: none"> • move method • move field • extract class • inline class • remove middle man • hide delegate 	<ul style="list-style-type: none"> • pull up field • push down field • pull up method • push down method • pull up constructor body • extract subclass/ superclass • collapse hierarchy • replace inheritance with delegation • replace delegation with inheritance 	<ul style="list-style-type: none"> • self encapsulate field • encapsulate field/collection • replace data value with object • replace array with object • replace magic number with symbolic constant 	<ul style="list-style-type: none"> • decompose conditional • consolidate conditional expression • replace conditional with polymorphism 	<ul style="list-style-type: none"> • rename method • preserve whole object • introduce parameter object • parameterize method

EXTRACT METHOD

problema: tengo código que puede ser agrupado junto

```
void printOwing() {
    printBanner();

    // Print details.
    System.out.println("name: " + name);
    System.out.println("amount: " + getOutstanding());
}
```

```
void printOwing() {
    printBanner();
    printDetails(getOutstanding());
}

void printDetails(double outstanding) {
    System.out.println("name: " + name);
    System.out.println("amount: " + outstanding);
}
```

INLINE METHOD

problema: cuando el cuerpo de un método es más obvio que el método mismo

solución: reemplazo las llamadas al método con el contenido del método y elimine el método en sí

```
class PizzaDelivery {
    // ...
    int getRating() {
        return moreThanFiveLateDeliveries() ? 2 : 1;
    }
    boolean moreThanFiveLateDeliveries() {
        return numberOfLateDeliveries > 5;
    }
}
```

```
class PizzaDelivery {
    // ...
    int getRating() {
        return numberOfLateDeliveries > 5 ? 2 : 1;
    }
}
```

REPLACE TEMP WITH QUERY

problema: coloco el resultado de una expresión en una variable local para posteriormente usarlo en el código. Solución: mover la expresión a un método separado y devuelvo el resultado, estaría consultando al método en vez de usar una variable

sirve para evitar métodos largos, para poder usar una expresión desde otros métodos. La solución es extraer la expresión de un método, reemplazar todas las referencias a la variable temporal por la expresión, el nuevo método luego puede ser usado en otros métodos

```
double calculateTotal() {
    double basePrice = quantity * itemPrice;
    if (basePrice > 1000) {
        return basePrice * 0.95;
    }
    else {
        return basePrice * 0.98;
    }
}
```

```
double calculateTotal() {
    if (basePrice() > 1000) {
        return basePrice() * 0.95;
    }
    else {
        return basePrice() * 0.98;
    }
}
double basePrice() {
    return quantity * itemPrice;
}
```

SPLIT TEMPORARY VARIABLE

problema: tengo una variable local que es usada para almacenar varios valores intermedios dentro del método

solución: usar diferentes variables para diferentes valores. Cada variable debe ser responsable de una cosa en particular

```
double temp = 2 * (height + width);
System.out.println(temp);
temp = height * width;
System.out.println(temp);
```

```
final double perimeter = 2 * (height + width);
System.out.println(perimeter);
final double area = height * width;
System.out.println(area);
```

REEMPLACE METHOD WITH METHOD OBJECT

problema: tengo un método largo en el cual las variables locales están entrelazadas

solución: transformar el método en una clase separada para que las variables locales se conviertan en campos de la clase. Luego puedo dividir el método en varios métodos dentro de la clase

```
class Order {
    // ...
    public double price() {
        double primaryBasePrice;
        double secondaryBasePrice;
        double tertiaryBasePrice;
        // Perform long computation.
    }
}
```

```
class Order {
    // ...
    public double price() {
        return new PriceCalculator(this).compute();
    }
}

class PriceCalculator {
    private double primaryBasePrice;
    private double secondaryBasePrice;
    private double tertiaryBasePrice;

    public PriceCalculator(Order order) {
        // Copy relevant information from the
        // order object.
    }

    public double compute() {
        // Perform long computation.
    }
}
```

SUBSTITUTE ALGORITHM

problema: quiero reemplazar en algoritmo con uno nuevo?

solución: reemplazar el cuerpo del método donde debe ir el nuevo algoritmo

```
class Order {
    // ...
    public double price() {
        double primaryBasePrice;
        double secondaryBasePrice;
        double tertiaryBasePrice;
        // Perform long computation.
    }
}
```

```
class Order {
    // ...
    public double price() {
        return new PriceCalculator(this).compute();
    }
}

class PriceCalculator {
    private double primaryBasePrice;
    private double secondaryBasePrice;
    private double tertiaryBasePrice;

    public PriceCalculator(Order order) {
        // Copy relevant information from the
        // order object.
    }

    public double compute() {
        // Perform long computation.
    }
}
```

MOVE METHOD

un método es usado más en otra clase que en su propia clase, es una mala asignación. Un método está usando o usará muchos servicios que están definidos en una clase diferente a la suya. La solución es mover el método a la clase donde están los servicios que usa y convierto el método original en un simple delegado

mecánica

- reviso las variables de instancia usadas por el método a mover
- reviso super y subclases
- creo un nuevo método en la clase correcta
- copio el código al nuevo método
- determino como referenciar al target desde source
- reemplazo método original por llamada al método en target
- decidir si remover el método original o mantenerlo como delegación



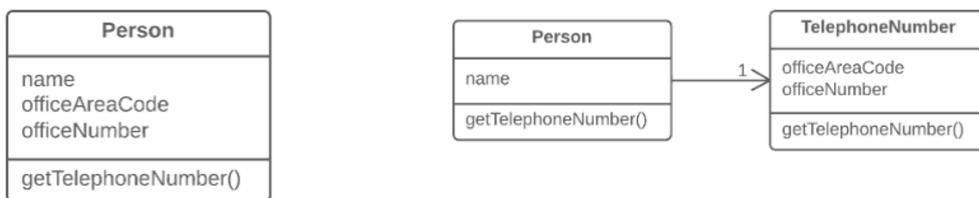
MOVE FIELD

problema: un campo es usado más en otra clase que en su propia clase
solución: creo un nuevo campo en la otra clase y redirijo todo al nuevo campo



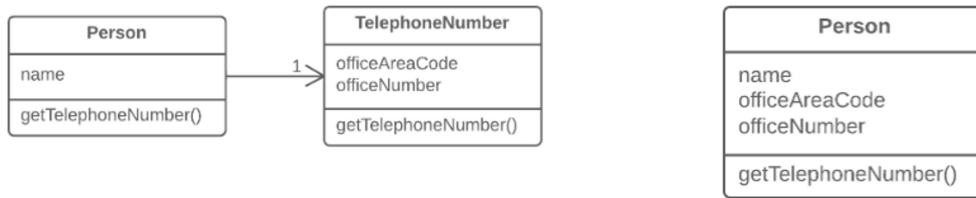
EXTRACT CLASS

problema: cuando una clase hace el trabajo de dos, solución: crear una nueva clase y acomodar la responsabilidad de los campos y métodos donde corresponda



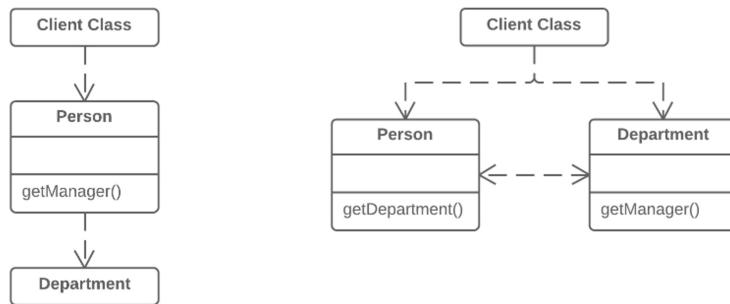
INLINE CLASS

problema: una clase no hace nada ni tampoco tiene responsabilidades, solución: mover todo a otra clase



REMOVE MIDDLE MAN

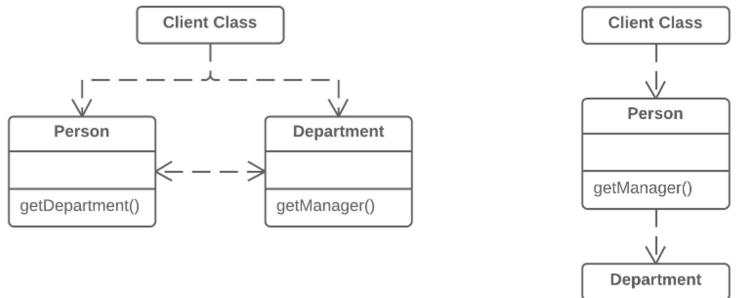
problema: una clase tiene muchos métodos que simplemente delegan a otros objetos, la solución es eliminar esos métodos y forzar al cliente a llamar directamente a los métodos finales



HIDE DELEGATE

problema: el cliente obtiene el objeto B de un campo o método del objeto A. Luego, el cliente llama a un método del objeto B

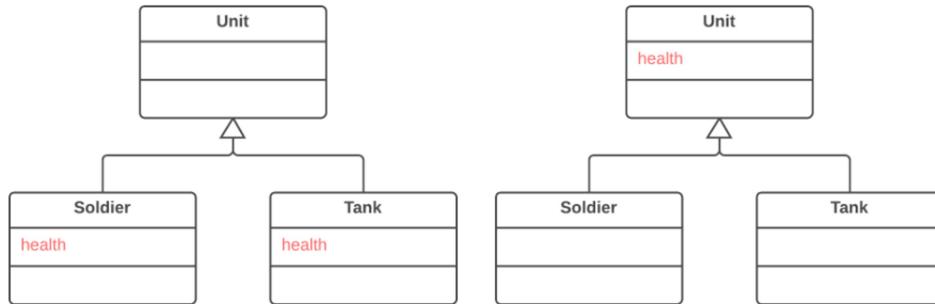
solución: crear un nuevo método en la clase A que delegue la llamada al objeto B. Ahora el cliente no conoce ni depende de la clase B



PULL UP FIELD

problema: dos clases tienen un mismo campo

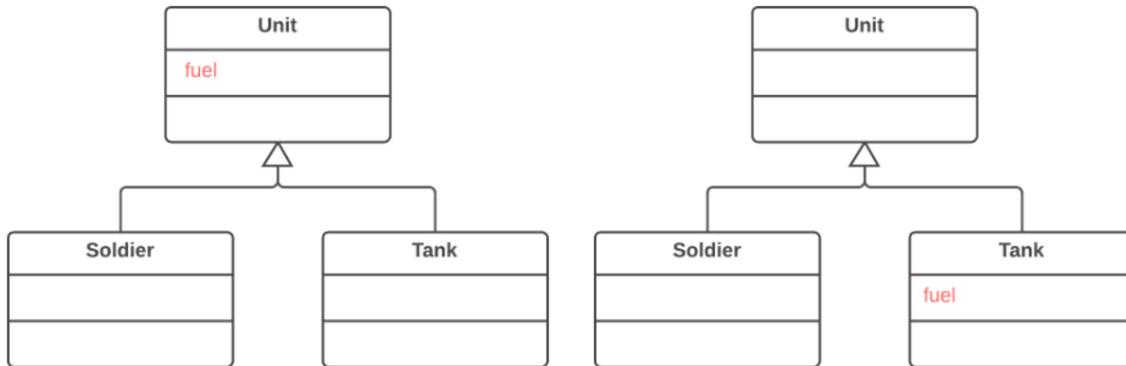
solución: sacar el campo de las subclases y moverlo a la superclase



PUSH DOWN FIELD

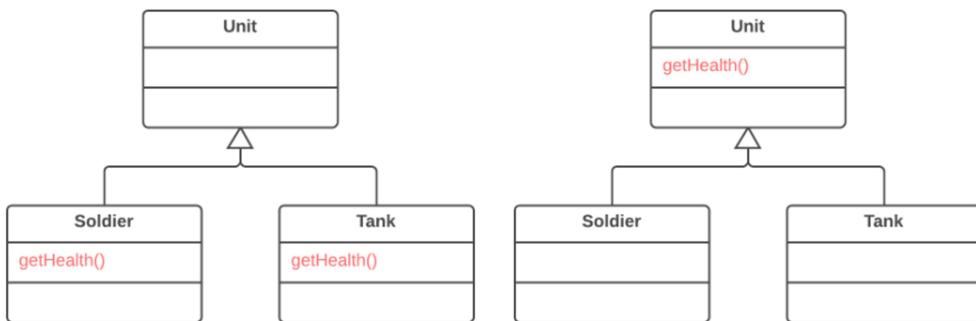
problema: el campo es usado en pocas subclases

solución: bajar el campo a las subclases que lo necesite



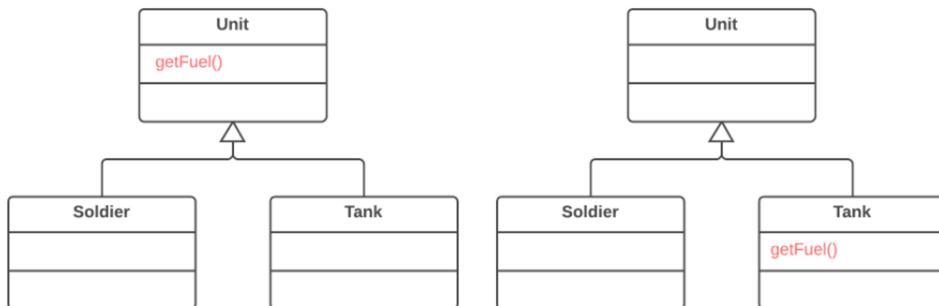
PULL UP METHOD

problema: tengo subclases que tienen métodos que hacen un trabajo similar. Solución: mover los métodos en común hacia una super clase



PUSH DOWN METHOD

problema: el comportamiento implementado en la superclase es usado por una sola subclase? entonces lo muevo a la subclase correspondiente



PULL UP CONSTRUCTOR BODY

problema: la subclase tiene constructores con código casi idéntico

solución: crea un constructor en la super clase y luego llamo al constructor de la superclase en las subclases

```

class Manager extends Employee {
    public Manager(String name, String id, int grade) {
        this.name = name;
        this.id = id;
        this.grade = grade;
    }
    // ...
}
  
```

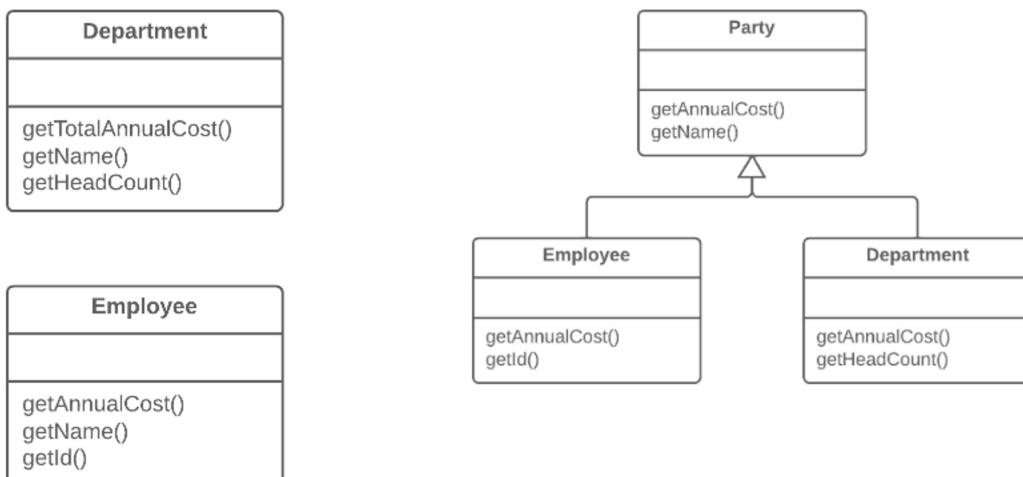
```

class Manager extends Employee {
    public Manager(String name, String id, int grade) {
        super(name, id);
        this.grade = grade;
    }
    // ...
}
  
```

EXTRACT SUPERCLASE

problema: tenés dos clases con campos y métodos comunes

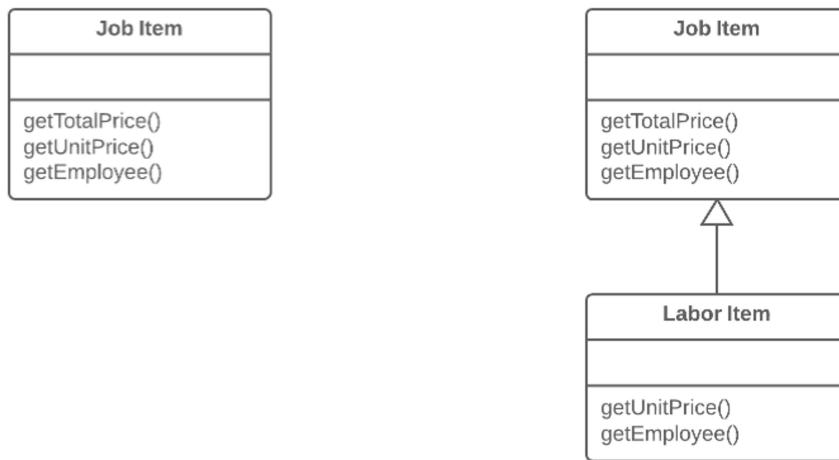
solución: crear una superclase compartida entre ellos y mover los campos y métodos en común



EXTRACT SUBCLASSE

problema: una tiene algunas características que se usan pocas veces

solución: crear una subclase y mover esas características



COLLAPSE HIERARCHY

problema: tengo una clase heredada la cual es lo mismo que su superclase

solución: unir superclase con subclase



SELF ENCAPSULATE FIELD

problema: accedes a los campos del objeto directamente sin utilizar getters/setters

solución: crear getters y setters para cada campo y usarlos para acceder a ellos

```

class Range {
    private int low, high;
    boolean includes(int arg) {
        return arg >= low && arg <= high;
    }
}
  
```

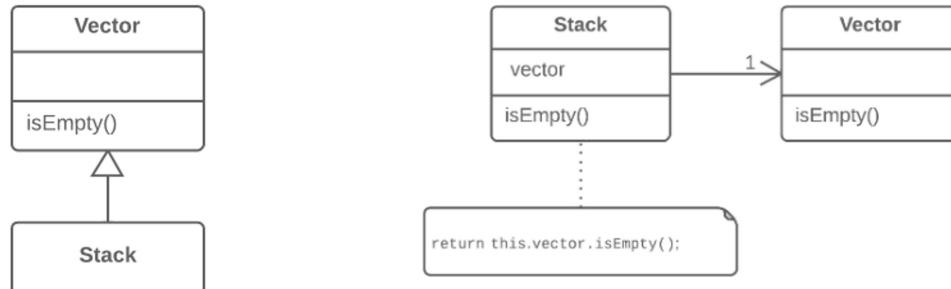
```

class Range {
    private int low, high;
    boolean includes(int arg) {
        return arg >= getLow() && arg <= getHigh();
    }
    int getLow() {
        return low;
    }
    int getHigh() {
        return high;
    }
}
  
```

REPLACE INHERITANCE WITH DELEGATION

problema: tenés una subclase que usa solo algunos métodos de su superclase

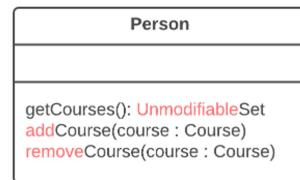
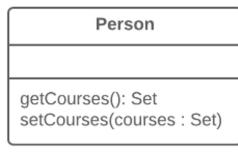
solución: crear un campo y poner el objeto de la superclase y delegar los métodos del objeto superclase y deshacerse de la herencia



ENCAPSULATE COLLECTION

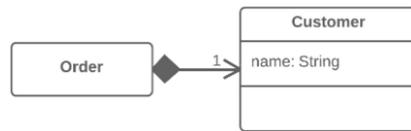
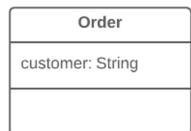
problema: una clase contiene una colección y un simple getter y setter para trabajar con la colección

solución: hacer que el valor de retorno por el método sea de solo lectura y crear métodos para agregar/eliminar elementos de la colección



REPLACE DATA VALUE WITH OBJECT

problema: una clase contiene un campo de datos el cual tiene comportamiento propio



REPLACE ARRAY WITH OBJECT

problema: tenés un arreglo que contiene varios tipos de datos

solución: reemplazar un arreglo con un objeto que separará los campos en cada elemento

```

String[] row = new String[2];
row[0] = "Liverpool";
row[1] = "15";
  
```

```

Performance row = new Performance();
row.setName("Liverpool");
row.setWins("15");
  
```

REPLACE MAGIC NUMBER WITH SYMBOLIC CONSTANT

problema: el código usa un número que tiene cierto significado en este

solución: reemplazar este numero con una constante que tiene un nombre que haga entender su propósito

```

double potentialEnergy(double mass, double height) {
    return mass * height * 9.81;
}
  
```

```

static final double GRAVITATIONAL_CONSTANT = 9.81;

double potentialEnergy(double mass, double height) {
    return mass * height * GRAVITATIONAL_CONSTANT;
}
  
```

DECOMPOSE CONDITIONAL

problema: tengo un condicional complejo con *if-then / else o switch*, solución: descomponerlo en métodos separados

```

if (date.before(SUMMER_START) || date.after(SUMMER_END))
    charge = quantity * winterRate + winterServiceCharge
}
else {
    charge = quantity * summerRate;
}
  
```

```

if (isSummer(date)) {
    charge = summerCharge(quantity);
}
else {
    charge = winterCharge(quantity);
}
  
```

CONSOLIDATE CONDITIONAL EXPRESSION

problema: tenés múltiples condicionales para lidar con la misma acción

solución: meternos en una sola expresión

```

double disabilityAmount() {
    if (seniority < 2) {
        return 0;
    }
    if (monthsDisabled > 12) {
        return 0;
    }
    if (isPartTime) {
        return 0;
    }
    // Compute the disability amount.
    // ...
}
  
```

```

double disabilityAmount() {
    if (isNotEligibleForDisability()) {
        return 0;
    }
    // Compute the disability amount.
    // ...
}
  
```

REPLACE CONDITIONAL WITH POLYMORPHISM

tengo un condicional que hace varias acciones según el tipo de objeto o propiedades. La solución es crear subclases que coincidan con las ramas del condicional, en ellos crear un método compartido y mover el código correspondiente. Luego, reemplazar el condicional con la llamada al método importante

mecánica

- crear una jerarquía de clases necesarias
- si el condicional es parte de un método largo, aplico extract method
- por cada subclase creo un método que sobrescribe al método que contiene el condicional, copio el código de la condición correspondiente en el método de la subclase, borro condición y código del método en la superclase
- hacer que el método en la superclase sea abstracto

```
class Bird {
    // ...
    double getSpeed() {
        switch (type) {
            case EUROPEAN:
                return getBaseSpeed();
            case AFRICAN:
                return getBaseSpeed() - getLoadFactor() * number;
            case NORWEGIAN_BLUE:
                return (isNailed) ? 0 : getBaseSpeed(voltage);
        }
        throw new RuntimeException("Should be unreachable");
    }
}
```

```
abstract class Bird {
    // ...
    abstract double getSpeed();
}

class European extends Bird {
    double getSpeed() {
        return getBaseSpeed();
    }
}

class African extends Bird {
    double getSpeed() {
        return getBaseSpeed() - getLoadFactor() * number;
    }
}

class NorwegianBlue extends Bird {
    double getSpeed() {
        return (isNailed) ? 0 : getBaseSpeed(voltage);
    }
}

// Somewhere in client code
speed = bird.getSpeed();
```

RENAME METHOD

problema: el nombre del método no explica lo que hace

solución: renombrarlo

REPLACE PARAMETER WITH METHOD

problema: llamar a un método y pasar sus resultados como parámetros de otro método mientras que ese método podría llamar a la consulta directamente, solución: en vez de pasar el valor a través de un parámetro, colocar una llamada de consulta dentro del cuerpo del método

```
int basePrice = quantity * itemPrice;
double seasonDiscount = this.getSeasonalDiscount();
double fees = this.getFees();
double finalPrice = discountedPrice(basePrice, seasonDiscount);
```

```
int basePrice = quantity * itemPrice;
double finalPrice = discountedPrice(basePrice);
```

PRESERVE WHOLE OBJECT

problema: tengo varios valores de un objeto luego tengo que pasarlo como parámetros para un método, solución: pasar el objeto entero

```
int low = daysTempRange.getLow();
int high = daysTempRange.getHigh();
boolean withinPlan = plan.withinRange(low, high);
```

```
boolean withinPlan = plan.withinRange(daysTempRange);
```

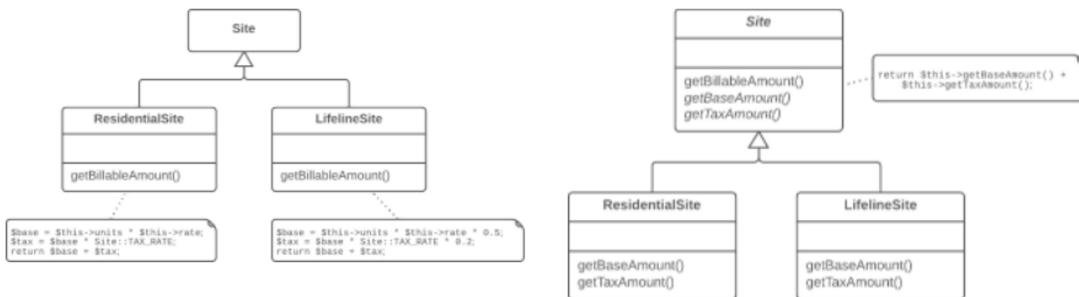
problema: el método contiene un repetido grupo de parámetros, solución: reemplazar los parámetros con un objeto

Customer
amountInvoicedIn (start : Date, end : Date)
amountReceivedIn (start : Date, end : Date)
amountOverdueIn (start : Date, end : Date)

Customer
amountInvoicedIn (date : DateRange)
amountReceivedIn (date : DateRange)
amountOverdueIn (date : DateRange)

FORM TEMPLATE METHOD

problema: la subclase implementa pasos que contienen pasos similares en el mismo orden, solución: mover la estructura de pasos a una superclase y dejar la implementación de diferentes pasos en las superclases. O sea aplico el template method pattern



REFACTORING Y TESTING

el testing sirve para detectar si un refactoring cambió el comportamiento y el refactoring puede romper un test aunque sea correcto

PERFORMANCE

la mejor manera de optimizar un programa, primero es escribir un programa bien factorizado y luego optimizarlo

REFACTORING TO PATTERNS

"los patrones de diseño proporcionan objetivos para sus refactorizaciones"

"los patrones están donde queres estar, las refactorizaciones son la forma de llegar a ese lugar"

la sobre-ingeniería significa construir software más sofisticado de lo que realmente necesita ser.

surge para acomodar futuros cambios pero como consecuencia el código sofisticado complica el mantenimiento, nadie lo entenderá

la poca ingeniería significa producir software con un diseño pobre

los patrones son tentadores para no quedarnos envueltos y no arrastrar un mal diseño pero nos puede llevar al otro extremo, por lo tanto es necesario conocer las consecuencias de un patrón

STATE OR STRATEGY?

el patrón State es útil para una clase que debe realizar transiciones entre estados fácilmente, mientras que el Strategy es útil para permitir que una clase delegue la ejecución de un algoritmo a una instancia de una familia de estrategias.

State	Strategy
se settea por el cliente, debe conocer las posibles estrategias concretas	
puede definir muchos mensajes	suele tener un único mensaje público
se conocen entre sí	no se conocen