

Patrones de diseño

Clase 1: 7/10 (strategy y factory)

Patrones de diseño de software: es una repetición de la solución a un problema determinado. Muchas personas resuelven un problema de una determinada forma → se transforma en patrón. Cada vez que tenemos un problema de X tipo, el patrón es una solución válida.

No solo es importante el patrón sino también el **contexto** en el cual se aplica el mismo. Lo importante es saber reconocer el patrón, por eso es necesario saber en qué contexto cada uno es una buena solución.

Vamos a poner foco en **reconocer el patrón**. ¿Cuándo se da qué cosa se da X solución?

Refactorizar: proceso por el cual se modifica el comportamiento interno del código sin modificar el funcionamiento del mismo.

CASO 1: STRATEGY

clase → loan (préstamo)

metodo → capital: devuelve un double

test → loan test: crea loans con distintos atributos y chequea que el capital esté bien calculado para cada caso.

```
public class LoanTest {
    @Test
    public void testCalculateCapital() {
        final double commitment = 1;
        final double outstanding = 0.0;
        final int riskRating = 2;
        final Date maturity = new Date();
        final Date expiry = new Date();

        Loan termLoan = new Loan(commitment, outstanding, riskRating, maturity, expiry:null);
        Loan revolverLoan = new Loan(commitment, outstanding, riskRating, maturity:null, expiry);
        Loan rctlLoan = new Loan(commitment, outstanding, riskRating, maturity, expiry);

        assertNotNull(termLoan);
        assertNotNull(revolverLoan);
        assertNotNull(rctlLoan);

        assertTrue(termLoan.capital() == 2.0);
        assertTrue(revolverLoan.capital() == 0.0);
        assertTrue(rctlLoan.capital() == 1.2);
    }
}
```

Observaciones del código:

- Un único test que no es atómico, es decir no hace una única cosa sino que testea diferentes casos.
- Como consecuencia de lo mismo el nombre del método no es muy descriptivo; cuanto más atómico, más específico lo podemos hacer.
- El constructor de la clase loan tiene muchos parámetros, podrían estar reducidos en un objeto.

Clase Loan:

```

5 public class Loan {
6     private static final long MILLIS_PER_DAY = 86400000;
7     private static final long DAYS_PER_YEAR = 365;
8
9     private final double commitment;
10    private final double outstanding;
11    private final int riskRating;
12    private final Date maturity;
13    private final Date expiry;
14
15    public Loan(final double commitment, double outstanding, int riskRating, Date maturity, Date expiry) {
16        this.commitment = commitment;
17        this.outstanding = outstanding;
18        this.riskRating = riskRating;
19        this.maturity = maturity;
20        this.expiry = expiry;
21    }
22
23    public double getCommitment() {
24        return commitment;
25    }
26
27    public double getOutstanding() {
28        return outstanding;
29    }
30
31    public int getRiskRating() {
32        return riskRating;
33    }
34
35    public Date getMaturity() {
36        return maturity;
37    }
38
39    public Date getExpiry() {
40        return expiry;

```

```

    public double capital() {
        if (this.expiry == null && this.maturity != null) {
            return this.commitment * this.duration() * this.riskFactor();
        }
        if (this.expiry != null && this.maturity == null) {
            if (this.getUnusedPercentage() != 1.0) {
                return this.commitment * this.getUnusedPercentage() * this.duration() * this.riskFactor();
            } else {
                return (this.outstandingRiskAmount() * duration() * this.riskFactor()) + (this.unusedRiskAmount() * this.duration() * this.riskFactor());
            }
        }
        return 1.2;
    }
}

```

Observaciones de código de **capital**:

- Si tenemos en cuenta también el código de test podemos interpretar que los ifs que se encuentran en este método tiene relación con el tipo de loan que se crea.
- No se está aprovechando el polimorfismo.

Lo que sí, se puede ver que hay una correlación entre la forma de calcular el capital y los distintos tipos de loan.

Posibles soluciones:

1. **Herencia:** Lo que podemos hacer es crear dos clases que parten de loan, por ejemplo long time loan/revolver loan y que cada una de ellas implemente su propio método capital (herencia). Esta solución puede ser interesante cuando dos métodos SIEMPRE van juntas, entonces creo una interfaz que involucre ambas cosas.
2. **Clase abstracta o Interfaz:** Podemos también crear una abstracción (como interfaz, clase abstracta, etc) de solo lo que me importa, en este caso de capital. La abstracción solo hace una única tarea.

Principio de Diseño presente en estas soluciones: **Principio segregación de interfaces:** agarrar una clase que implementa varias cosas y dividirla en varias que implementan una sola cosa

Avance del código:

```
public double capital() {
    if (this.expiry == null && this.maturity != null) {
        return this.termCapital();
    }
    if (this.expiry != null && this.maturity == null) {
        return this.revolverCapital();
    }
    return this.rctlCapital();
}

private double termCapital() {
    return this.commitment * this.duration() * this.riskFactor();
}

private double revolverCapital() {
    if (this.getUnusedPercentage() != 1.0) {
        return this.commitment * this.getUnusedPercentage() * this.duration() * this.riskFactor();
    } else {
        return (this.outstandingRiskAmount() * duration() * this.riskFactor()) + (this.unusedRiskAmount() * this.duration() *
    }
}
```

Observaciones del código:

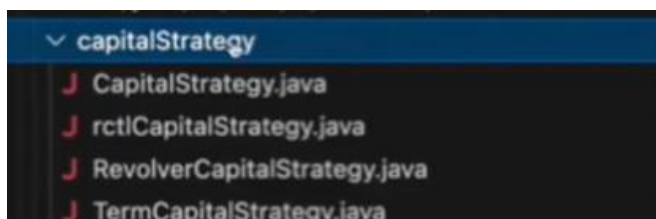
- Tengo métodos **privados** para cada tipo de loan para poder calcular el capital en cada caso particular.
- Si aparece un nuevo tipo de loan que implementa capital de una forma diferente, el código no es tan extensible (hay que agregar otro if al método lo cual no es lo ideal)

Avance de código:

Como vimos la relación entre los ifs de capital y el tipo de Loan vamos a devolver ese tipo específico.

Entonces ahora que tenemos esos tres métodos, podríamos abstraerlos y pasar la abstracción a la función capital, y los parámetros que es lo que devuelve.

Creo un capitalStrategy.java y cada tipo de loan con su implementación del método capital. Esto forma parte del patrón **STRATEGY**, componemos un objeto que tiene una estrategia de cómo calcular el capital (en este caso)



```
5 public interface CapitalStrategy {
6     public double capital(final Loan loan);
7 }
8
```

Acá tenemos la interfaz, con su método, y según el tipo de loan lo vamos a sobrecribir:

```
5 public class rctlCapitalStrategy implements CapitalStrategy {
6
7     @Override
8     public double capital(final Loan loan) {
9         return 1.2;
10    }
11
12 }
```

```
5 public class TermCapitalStrategy implements CapitalStrategy {
6
7     @Override
8     public double capital(final Loan loan) {
9         return loan.getCommitment() * loan.duration() * loan.riskFactor();
10    }
11
12 }
```

```

5 public class RevolverCapitalStrategy implements CapitalStrategy {
6
7     @Override
8     public double capital(final Loan loan) {
9         if (loan.getUnusedPercentage() != 1.0) {
10             return loan.getCommitment() * loan.getUnusedPercentage() * loan.duration() * loan.riskFactor();
11         } else {
12             return (loan.outstandingRiskAmount() * loan.duration() * loan.riskFactor()) + (loan.unusedRiskAmount() * loan.duration() * loan.riskFactor());
13         }
14     }
15 }
16

```

Otra cosa que tenemos que modificar es el método capital() que nos quedó. Nos quedaron todos los ifs para una instancia diferente en cada caso. Como lo que chequean los ifs son parámetros del constructor, lo que podríamos hacer es hacer el chequeo en el constructor y dependiendo la condición, devuelvo una instancia de Strategy diferente:

```

public Loan(final double commitment, double outstanding, int riskRating, Date maturity, Date expiry) {
    this.commitment = commitment;
    this.outstanding = outstanding;
    this.riskRating = riskRating;
    this.maturity = maturity;
    this.expiry = expiry;

    if (this.expiry == null && this.maturity != null) {
        this.capitalStrategy = new TermCapitalStrategy();
    } else if (this.expiry != null && this.maturity == null) {
        this.capitalStrategy = new RevolverCapitalStrategy();
    } else {
        this.capitalStrategy = new RCTLCapitalStrategy();
    }
}

public double capital() {
    return this.capitalStrategy.capital(this);
}

```

Ahora esta solución no es definitiva porque si tengo un nuevo tipo de Loan tengo que tocar el constructor y agregar más condiciones.

Pero... con estos cambios, pasamos a tener un problema de **Construcción**.

Observaciones de código:

- El método capital ahora está clausurado.
- Ahora tengo los ifs en el constructor, es decir que tengo el mismo tema de que tengo que agregar un if en caso de agregar un nuevo tipo de loan. Ahora el problema es un problema de construcción (si quiero puedo moverlo al main del sistema para tomar la decisión ahí mismo de qué estrategia voy a usar).
- Si yo quiero que el modo de calcular el capital cambie dinámicamente, la herencia no sirve.

Podría eliminar la lógica de los ifs del constructor y mandarlo al main del sistema o algo así. Lo que puedo hacer es utilizar **Composición**, para componer Loan con un atributo, como capitalStrategy, que tenga ciertas características (expiry y maturity por ejemplo, y se instancia en cada caso de manera diferente).

Para poder descartar este if, se lo delegamos a una Interfaz que se encargue de construir estas estrategias.

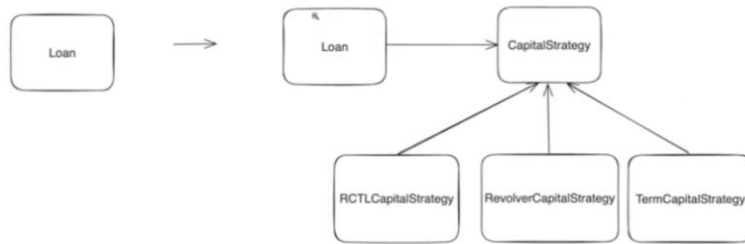


Diagrama para entender desde el diseño como evolucionó el código.

Una opción para eliminar los ifs en el constructor podría tener una clase factory la cual se encargue de hacer los instanciamientos necesarios. Los ifs quedan encapsulados y el problema ahora es creacional. Cuando viene una nueva estrategia solo toca al factory.

```
public Loan(final double commitment, double outstanding, int riskRating, Date maturity, Date expiry) {
    this.commitment = commitment;
    this.outstanding = outstanding;
    this.riskRating = riskRating;
    this.maturity = maturity;
    this.expiry = expiry;

    this.capitalStrategy = LoanCapitalFactory.createCapitalStrategy(...);
}
```

Otra opción es tener la estrategia inyectada en los parámetros del constructor. La conclusión es que el if se extrae a una lógica creacional; se puede inyectar la factory, o se puede usar Singleton por ejemplo.

CASO 2: FACTORY

Tomamos la misma clase, pero ahora vamos a analizar el constructor:

Avance de código:

```
public class LoanTest {
    @Test
    public void testCreateTermLoan() {
        final double commitment = 1;
        final int riskRating = 2;
        final Date maturity = new Date();
        double commitment = ar.uba.fi.tecnicas.exercisel.LoanTest.testCreateTermLoan()

        Loan termLoan = new Loan(commitment, riskRating, maturity);

        assertNotNull(termLoan);
    }
}
```

Ahora el test pasa solo esos valores al constructor para construir ese tipo de loan (termLoan) y ese va a otros constructores

```
public Loan(final double commitment, int riskRating, Date maturity) {
    this(commitment, outstanding:0.00, riskRating, maturity, expiry:null);
}

public Loan(final double commitment, int riskRating, Date maturity, Date expiry) {
    this(commitment, outstanding:0.00, riskRating, maturity, expiry);
}

public Loan(final double commitment, double outstanding, int riskRating, Date maturity, Date expiry) {
    this(capitalStrategy:null, commitment, outstanding, riskRating, maturity, expiry);
}

public Loan(final CapitalStrategy capitalStrategy, final double commitment, int riskRating, Date maturity, Date expiry) {
    this(capitalStrategy, commitment, outstanding:0.00, riskRating, maturity, expiry);
}
```

Vemos que ahora tenemos varios constructores, y en algunos le estamos pasando la capitalStrategy por parámetro

Básicamente *Reemplazamos varios ifs por algunos constructores.*

```
public Loan(final CapitalStrategy capitalStrategy, final double commitment, double outstanding, int riskRating, Date maturity) {
    this.commitment = commitment;
    this.outstanding = outstanding;
    this.riskRating = riskRating;
    this.maturity = maturity;
    this.expiry = expiry;
    this.capitalStrategy = capitalStrategy;

    if (capitalStrategy == null) {
        if (expiry == null) {
            this.capitalStrategy = new CapitalStrategyTermLoan();
        } else if (maturity == null) {
            this.capitalStrategy = new CapitalStrategyRevolver();
        } else {
            this.capitalStrategy = new CapitalStrategyRCTL();
        }
    }
}
```

Defecto:

```
14
15 Loan termLoan = new Loan(commitment, riskRating, maturity);
16
```

Yo llamo a un constructor de Loan, y yo como programador me tengo que acordar de que si quiero un termLoan tengo que mandarle esos parámetros. Sería mejor poder llamar a un new TermLoan(...)

Observaciones de código:

- Sigo teniendo el problema de que no hay nada en el código que diga que es un termLoan (por ejemplo), tengo que saber que parámetros pasarle para crear cada tipo de loan.
- Una opción podría ser ir por herencia.
- Puedo definir un método estático de la clase que devuelva un new termLoan:
Podríamos ir por la herencia, o llamar a un *método estático propio de la clase que se llame newTermLoan*, y lo llamas como *Loan.newTermLoan(...)*, que adentro tenga un constructor de un term loan.

Avance de código:

```
public class LoanTest {
    @Test
    public void testCreateTermLoan() {
        final double commitment = 1;
        final int riskRating = 2;
        final Date maturity = new Date();

        Loan termLoan = Loan.createTerm(commitment, riskRating, maturity);

        assertNotNull(termLoan);
    }
}
```

Ahora tengo un constructor específico para el tipo de loan.

Se puede hacer lo mismo para todos los tipos.

```
public static Loan createTerm(final double commitment, int riskRating, Date maturity) {
    return new Loan(capitalStrategy:null, commitment, outstanding:0.00, riskRating, maturity, expiry:null);
}
```


Ahora el constructor te dice que necesitas para cada tipo de loan y no es algo que te tenes que acordar vos. El código me dice explícitamente que tipo de Loan estoy creando sin tener que acordarme yo.

```
public static Loan createTerm(final double commitment, int riskRating, Date maturity) {  
    return new Loan(capitalStrategy:null, commitment, outstanding:0.00, riskRating, maturity, expiry:null);  
}  
  
public static Loan createTerm(final CapitalStrategy capitalStrategy, final double commitment, double outstanding, int riskRating, Date maturity, Date expiry) {  
    return new Loan(capitalStrategy, commitment, outstanding, riskRating, maturity, expiry);  
}  
  
public static Loan createRevolver(final double commitment, double outstanding, int riskRating, Date expiry) {  
    return new Loan(capitalStrategy:null, commitment, outstanding, riskRating, maturity:null, expiry);  
}  
  
public static Loan createRevolver(final CapitalStrategy capitalStrategy, final double commitment, double outstanding, int riskRating, Date expiry) {  
    return new Loan(capitalStrategy, commitment, outstanding, riskRating, maturity:null, expiry);  
}  
  
public static Loan createRCTL(final double commitment, double outstanding, int riskRating, Date maturity, Date expiry) {  
    return new Loan(capitalStrategy:null, commitment, outstanding, riskRating, maturity, expiry);  
}  
  
public static Loan createRCTL(final CapitalStrategy capitalStrategy, final double commitment, double outstanding, int riskRating, Date maturity, Date expiry) {  
    return new Loan(capitalStrategy, commitment, outstanding, riskRating, maturity, expiry);  
}
```

Podríamos decir que son “creadores” (osea como un constructor, pero no la palabra constructor literal de Java, pero en definitiva estoy creando objetos).

Son métodos cuya **única responsabilidad es crear un objeto**, es parte del patrón **FACTORY METHOD**. Son fábricas de objetos. En general el mismo se ve en una jerarquía de clases, hay un método que es la creación de un objeto que se encuentra en lo más alto de la jerarquía, las que extienden de las mismas lo redefinen para crear un objeto u otro.

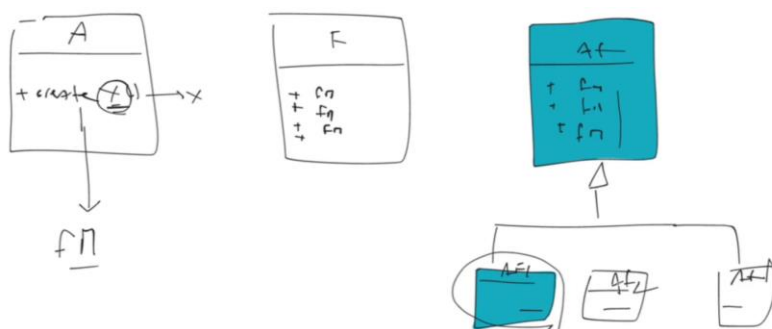
=> una factory es una clase que **SOLO** tiene factory methods.

Entonces una FACTORY, seria una clase que **únicamente crea objetos**. Luego existe el concepto de Abstract Factory, que lo que agrega es un concepto de jerarquía. Tendría por ejemplo una factory de Loans y después una factory de ArgentinaLoans, UsaFactory, etc;

El concepto de abstract factory donde se tiene una familia de factory con una jerarquía de fábricas. Es el simple hecho de abstraer las fábricas.

Diagrama de Factory:

Tengo una clase A con un método createX y me devuelve un x. Ese create, es un **factory method**. Luego, el concepto de Factory, es que **todos sus métodos son factory methods**. Por otro lado, la Abstract Factory tiene la definición de los factory methods, y cada factory de cada “familia” los sobrescribe.



La Abstract Factory se puede implementar como una Interfaz, o como una Clase Abstracta; depende si quiero o no tener algunos métodos comunes no concretos.

Volviendo al ejemplo... Avance de código:

Todavía tengo el constructor de Loan como método público, entonces tiene la posibilidad de crear como el programador quiere el loan => lo convertimos en privado y solo se puede acceder a new Loan desde el creador de cada tipo de loan. La idea es usar las herramientas del lenguaje para que el código se comporte como queremos.

```
6 public class LoanFactory {
7     public static Loan createTerm(final double commitment, int riskRating, Date maturity) {
8         return new Loan(capitalStrategy:null, commitment, outstanding:0.00, riskRating, maturity, expiry:null);
9     }
10
11
12     public static Loan createTerm(final CapitalStrategy capitalStrategy, final double commitment, double outstanding, int riskRating, Date maturity, Date expiry) {
13         return new Loan(capitalStrategy, commitment, outstanding, riskRating, maturity, expiry);
14     }
15
16     public static Loan createRevolver(final double commitment, double outstanding, int riskRating, Date expiry) {
17         return new Loan(capitalStrategy:null, commitment, outstanding, riskRating, maturity:null, expiry);
18     }
19
20     public static Loan createRevolver(final CapitalStrategy capitalStrategy, final double commitment, double outstanding, int riskRating, Date expiry) {
21         return new Loan(capitalStrategy, commitment, outstanding, riskRating, maturity:null, expiry);
22     }
23
24     public static Loan createRCTL(final double commitment, double outstanding, int riskRating, Date maturity, Date expiry) {
25         return new Loan(capitalStrategy:null, commitment, outstanding, riskRating, maturity, expiry);
26     }
27
28     public static Loan createRCTL(final CapitalStrategy capitalStrategy, final double commitment, double outstanding, int riskRating, Date maturity, Date expiry) {
29         return new Loan(capitalStrategy, commitment, outstanding, riskRating, maturity, expiry);
30     }
31 }
32
```

Además, creo una clase factory con todos los métodos construcción para pasar todos los create a una LoanFactory.

```
16 private CapitalStrategy capitalStrategy;
17
18 Loan(final CapitalStrategy capitalStrategy, final double commitment, double outstanding, int riskRating, Date maturity, Date expiry) {
19     this.commitment = commitment;
20     this.outstanding = outstanding;
21     this.riskRating = riskRating;
22     this.maturity = maturity;
23     this.expiry = expiry;
24     this.capitalStrategy = capitalStrategy;
25
26     if (capitalStrategy == null) {
27         if (expiry == null) {
28             this.capitalStrategy = new CapitalStrategyTermLoan();
29         } else if (maturity == null) {
30             this.capitalStrategy = new CapitalStrategyRevolver();
31         } else {
32             this.capitalStrategy = new CapitalStrategyRCTL();
33         }
34     }
35 }
```

En el método de Loan tengo que sacar el private ya que sino desde la clase factory no podría acceder para hacer la creación de cada tipo de loan. Ahora tiene visibilidad de tipo package, es decir, que el constructor sólo es visible para todo lo que está en el mismo paquete => metiendo la factory y loan dentro del mismo paquete solo factory puede hacer uno del new Loan.

También puedo mover la lógica de ifs a la fábrica ya que eso es lógica de creación.

Ej: si te da la estrategia null tira error y sino crea con lo necesario.

```
19 public static Loan createTerm(final CapitalStrategy capitalStrategy, final double commitment, double outstanding, int riskRating, Date maturity, Date expiry) {
20     if (capitalStrategy == null) {
21         throw new NullPointerException(s:"Do not support null capital strategy");
22     }
23     return new Loan(capitalStrategy, commitment, outstanding, riskRating, maturity, expiry:null);
24 }
```

En caso de no recibir el parámetro hace el new correspondiente.

```
public static Loan createRevolver(final double commitment, double outstanding, int riskRating, Date expiry) {
    return new Loan(new CapitalStrategyRevolver(), commitment, outstanding, riskRating, maturity:null, expiry);
}
```

CLASE 2:

CASO 1: PATRON BUILDER

Partamos del siguiente problema:

Se desea permitir a los clientes de una pizzería:

- Ordenar pizzas de diferentes tamaños.
- Seleccionar diferentes combinaciones de ingredientes, pero no necesariamente todos.

Tendría un constructor principal así:

```
problem.java
Pizza(int size, boolean cheese, boolean pepperoni, boolean bacon, ...) { ... }
```

Pero el problema es que tengo un constructor gigante, y un montón de parámetros que no se van a terminar usando, por ejemplo si solo le quiero poner queso, voy a tener un montón de nules al pedo.

Otra solución sería sobrecargar el constructor; tener varios constructores para una clase, cada uno diferente:

```
problem.java
Pizza(int size) { ... }
Pizza(int size, boolean cheese) { ... }
Pizza(int size, boolean cheese, boolean pepperoni) { ... }
Pizza(int size, boolean cheese, boolean pepperoni, boolean bacon) { ... }
```

Pero una vez más... esto complejiza mucho mi clase.

SOLUCION: Patron Builder

Para poder implementarla, una solución sería implementar el builder como una clase dentro de la misma clase Pizza:

```

1 package org.example;
2
3 public class Pizza {
4     private int size;
5     private boolean cheese;
6     private boolean pepperoni;
7     private boolean bacon;
8
9     private Pizza(Builder builder) {
10         this.size = builder.size;
11         this.cheese = builder.cheese;
12         this.pepperoni = builder.pepperoni;
13         this.bacon = builder.bacon;
14     }
15
16     public String toString() {
17         return "Pizza{" +
18             "size=" + size +
19             ", cheese=" + cheese +
20             ", pepperoni=" + pepperoni +
21             ", bacon=" + bacon +
22             '}';
23     }
24
25     public static class Builder {
26         private int size;
27         private boolean cheese = false;
28         private boolean pepperoni = false;
29         private boolean bacon = false;
30
31         public Builder(int size) {
32             this.size = size;

```

```

public static class Builder {
    private int size;
    private boolean cheese = false;
    private boolean pepperoni = false;
    private boolean bacon = false;

    public Builder(int size) {
        this.size = size;
    }

    public Builder cheese() {
        cheese = true;
        return this;
    }

    public Builder pepperoni() {
        pepperoni = true;
        return this;
    }

    public Builder bacon() {
        bacon = true;
        return this;
    }

    public Pizza build() {
        return new Pizza(this);
    }
}

```

Tenemos la clase, con todos sus parámetros, y vemos que ahora el constructor de Pizza recibe un Builder, en vez de todos los parámetros. Por otro lado notamos que ese constructor es **privado**; no queremos llamarlo desde otro lado que no sea el Builder.

El Builder sí es public, y tiene algunos parámetros *opcionales*, que son los que están seguidos de = false.

Los métodos seguidos al Builder settea los parámetros del builder para poder ir agregando lo que se necesita.

Por último, tenemos el método **build()** que lo que va a hacer es devolvernos lo que queremos (la Pizza), pasándole como parámetro el builder.

Dependiendo del tamaño de la clase Pizza, podríamos querer poner el constructor adentro o afuera; capaz si la clase es muy grande el constructor “ensucia” a la clase, y lo podría poner afuera como un PizzaBuilder.

Acá tenemos un ejemplo de cómo se construye una pizza:

```

class PizzaTest {
    @Test
    void testPizza() {
        Pizza pizza = new Pizza.Builder(size 12)
            .cheese()
            .pepperoni()
            .bacon()
            .build();
        assertEquals("expected: 'Pizza{size=12, cheese=true, pepperoni=true, bacon=true}', pizza.toString());
    }
}

```

Preferencia del Builder vs los Setters

El Builder me separa la construcción del objeto construido. Con los setters, hasta no asignar todos los parámetros, el objeto queda en un estado inconsistente; no está bueno tener una pizza que esté a medio hacer, y que se intente usar la pizza por ejemplo, antes de que se terminen de setear todos los parámetros.

Esto puede generar problemas en ambientes multithread.

Builder vs Factory

Lo más destacable del Builder es la **Flexibilidad**; para el caso de los múltiples parámetros, y problemas de un “paso a paso” el builder puede ser más conveniente.

Factory

Busca abstraer el proceso de creación de diferentes tipos de objeto.

El foco es *que objeto crear*.

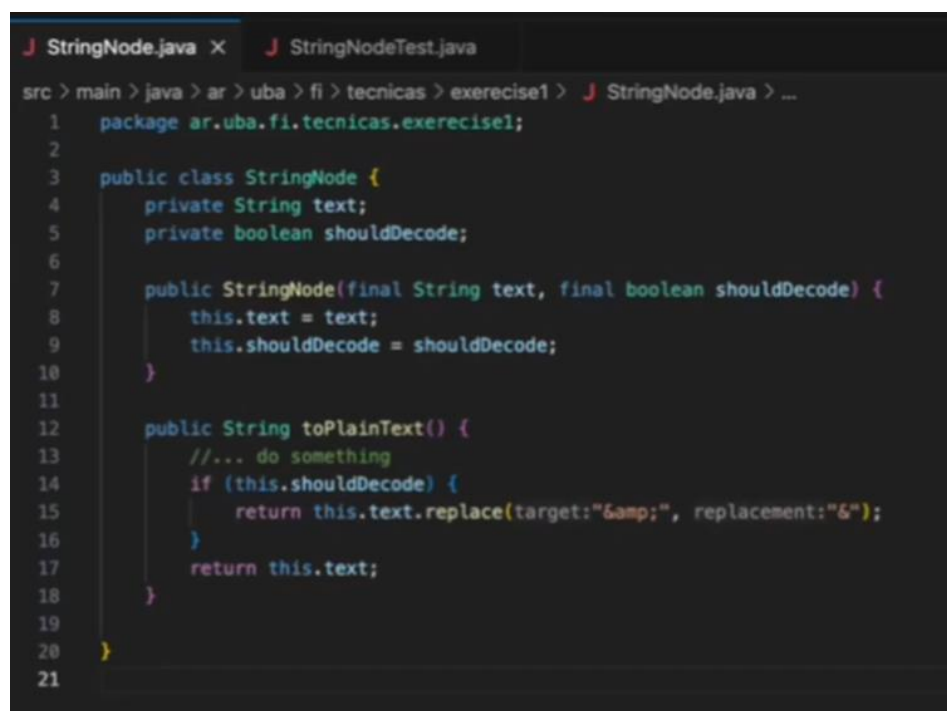
Builder

Busca resolver múltiples opciones y parámetros opcionales.

El foco es *como crear el objeto de forma flexible y paso a paso*.

CASO 2: PATRON DECORATOR

Tenemos una clase StringNode que tiene un texto y si se tiene que decodificar, se reemplazan algunos caracteres



```
StringNode.java X StringNodeTest.java
src > main > java > ar > uba > fi > tecnicas > exercise1 > StringNode.java > ...
1 package ar.uba.fi.tecnicas.exercise1;
2
3 public class StringNode {
4     private String text;
5     private boolean shouldDecode;
6
7     public StringNode(final String text, final boolean shouldDecode) {
8         this.text = text;
9         this.shouldDecode = shouldDecode;
10    }
11
12    public String toPlainText() {
13        //... do something
14        if (this.shouldDecode) {
15            return this.text.replace(target:"&", replacement:"&");
16        }
17        return this.text;
18    }
19
20 }
21
```

Acá está la clase Test:

```

StringNode.java X StringNodeTest.java
src > main > java > ar > uba > fi > tecnicas > ejercicio1 > StringNode.java > ...
1 package ar.uba.fi.tecnicas.ejercicio1;
2
3 public class StringNode {
4     private String text;
5     private boolean shouldDecode;
6
7     public StringNode(final String text, final boolean shouldDecode) {
8         this.text = text;
9         this.shouldDecode = shouldDecode;
10    }
11
12    public String toPlainText() {
13        //... do something
14        if (this.shouldDecode) {
15            return this.text.replace(target:"&","&");
16        }
17        return this.text;
18    }
19
20 }
21

```

Acá vemos una solución, de dividir responsabilidades:

```

7 import ar.uba.fi.tecnicas.ejercicio1.DecodedStringNode;
8 import ar.uba.fi.tecnicas.ejercicio1.SimpleStringNode;
9 import ar.uba.fi.tecnicas.ejercicio1.StringNode;
10
11 public class StringNodeTest {
12     @Test
13     public void testToPlainTextShouldDecode() {
14         String value = "This is a test & encoded";
15         String RESULT = "This is a test & encoded";
16
17         StringNode node = new DecodedStringNode(new SimpleStringNode(value));
18         String expected = node.toPlainText();
19
20         assertEquals(expected, RESULT);
21     }
22 }
23

```

Ahora el DecodedStringNode llama a un SimpleStringNode, donde tengo una abstracción (StringNode) y varios decoradores.

La abstracción:

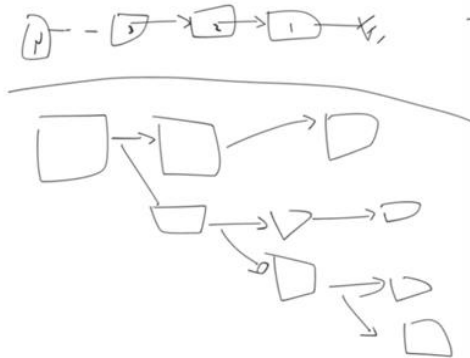
```

1 package ar.uba.fi.tecnicas.ejercicio1;
2
3 public interface StringNode {
4     public String toPlainText();
5 }
6

```

Se van enlazando las llamadas, hasta llegar al final de la lista y aplicar las operaciones. Cada clase hace algo más que la clase que la compone; hace algo más; extiende el servicio. Se puede ver como una lista simplemente enlazada.

Decorator vs Composite:



El decorator es como una lista enlazada de nodos, cada uno con una operación, mientras que el composite es como un árbol, donde cada uno puede componerse de uno o más nodos.

CASO 3: PATRON SINGLETON

Tenemos un Workshop Test con dos ctes con Jsons.

```
WorkshopTest.java X
src > test > java > ar > uba > fi > tecnicas > exercise1 > J WorkshopTest.java > {} ar.uba.fi.tecnicas.exercise1

1 package ar.uba.fi.tecnicas.exercise1;
2
3 import static org.junit.Assert.assertEquals;
4 import java.util.HashMap;
5 import java.util.Map;
6
7 import org.junit.*;
8 import ar.uba.fi.tecnicas
9 import ar.uba.fi.tecnicas.exercise1.CatalogApp;
10
11 public class WorkshopTest {
12     private static final Object WORKSHOP1_RESPONSE = "{\n\"id\":0,\n\"name\":\n\"Workshop1\", \n\"sections\": [\n\"Section1\", \n\"Section2\"]}";
13     private static final Object WORKSHOPS_RESPONSE = "[{\n\"id\":0,\n\"name\":\n\"Workshop1\", \n\"sections\": [\n\"Section1\", \n\"Section2\"]}, {
14
15     @Test
16     public void testHandleCreateWorkshop() {
17         CatalogApp app = new CatalogApp();
18
19         Map<String, String> parameters = new HashMap<>();
20         parameters.put(key:"name", value:"Workshop1");
21         parameters.put(key:"sections", value:"Section1,Section2");
22
23         String response = app.executeActionAndGetResponse(actionName:"new", parameters);
24
25         assertEquals(WORKSHOP1_RESPONSE, response);
26     }
27
28     @Test
29     public void testHandleGetAllWorkshops() {
30         CatalogApp app = new CatalogApp();
31
32         Map<String, String> parameters1 = new HashMap<>();
33         parameters1.put(key:"name", value:"Workshop1");
34         parameters1.put(key:"sections", value:"Section1,Section2");
35
36         app.executeActionAndGetResponse(actionName:"new", parameters1);
37
38         Map<String, String> parameters2 = new HashMap<>();
39         parameters2.put(key:"name", value:"Workshop2");
```

Y luego tenemos dos métodos donde, en el 1ro creamos el catalogo, y en la segunda ejecutamos y obtenemos la respuesta:

Esta es la CatalogApp:


```
J WorkshopTest.java  J CatalogApp.java X
src > main > java > ar > uba > fi > tecnicas > exercise1 > J CatalogApp.java > CatalogApp > NEW_WORKSHOP
1  package ar.uba.fi.tecnicas.exercisel;
2
3  import java.util.Arrays;
4  import java.util.Collection;
5  import java.util.List;
6  import java.util.Map;
7
8  public class CatalogApp {
9      private static final Object NEW_WORKSHOP = "new";
10     private static final Object ALL_WORKSHOPS = "all";
11
12     private WorkshopRepository workshopRepository = new WorkshopRepository();
13     private WorkshopSerializer workshopSerializer = new WorkshopSerializer();
14     private int workshopCounter = 0;
15
16     public String executeActionAndGetResponse(final String actionName, Map<String, String> parameters) {
17         if (actionName.equals(NEW_WORKSHOP)) {
18             int workshopId = workshopCounter++;
19             String workshopName = parameters.get(key:"name");
20             List<String> workshopSections = Arrays.asList(parameters.get(key:"sections").split(regex:""));
21             Workshop workshop = new Workshop(workshopId, workshopName, workshopSections);
22             workshopRepository.add(workshop);
23             return this.workshopSerializer.toJson(workshop);
24         } else if (actionName.equals(ALL_WORKSHOPS)) {
25             Collection<Workshop> workshops = this.workshopRepository.getAll();
26             return this.workshopSerializer.toJson(workshops);
27         } else {
28             return "default";
29         }
30     }
31 }
32
```

Se fija que acción es, obtiene del repositorio algunos workshops, o los crea, y los pasa a Json

Este es el WorkshopRepository:

```
J WorkshopTest.java  J CatalogApp.java  J Workshop.java  J WorkshopRepository.java X
src > main > java > ar > uba > fi > tecnicas > exercise1 > J WorkshopRepository.java > WorkshopRepository > add(Workshop)
1  package ar.uba.fi.tecnicas.exercisel;
2
3  import java.util.Collection;
4  import java.util.HashMap;
5  import java.util.Map;
6
7  public class WorkshopRepository {
8      private Map<String, Workshop> workshops = new HashMap<>();
9
10     public WorkshopRepository() {
11     }
12
13     public void add(final Workshop workshop) {
14         this.workshops.put(workshop.getName(), workshop);
15     }
16
17     public Collection<Workshop> getAll() {
18         return this.workshops.values();
19     }
20 }
21
```

Donde agrego nuevos repositorios y también puedo obtener todos.

En esta solución, estoy violando OCP, en el if de CatalogApp. Si yo quisiera obtener una nueva combinación de WORKSHOP, tengo que tocar ese if.

Si yo quiero agregar una nueva acción, debería agregar un else if, para atajar un nuevo caso. Para poder hacerlo más dinámico y para respetar OCP, una opción es en vez de pasarle una actionName, pasarle una **estrategia**, con sus parámetros, t cada estrategia maneja la devolución y sus parámetros de manera diferente.

Otra opción: Si yo quisiera mantener la firma, y querer pasar el actionName y parameters, me podría guardar las estrategias en un mapa, y luego según el actionName buscar en ese mapa la estrategia.

En cualquiera de los dos casos, necesito una **abstracción de la Strategy**.

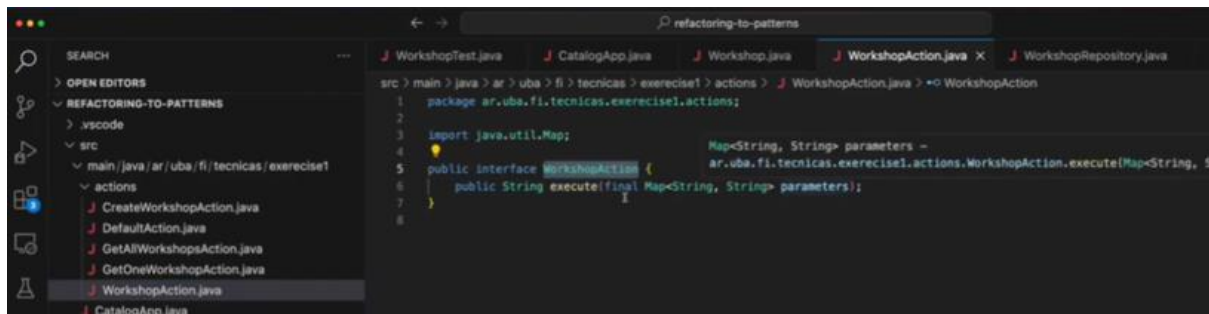
Si yo abstraigo la lógica de cada caso en una función:

```
16
17 public String executeActionAndGetResponse(final String actionName, Map<String, String> parameters) {
18     if (actionName.equals(NEW_WORKSHOP)) {
19         return this.executeCreateWorkshopAction(parameters);
20     } else if (actionName.equals(ALL_WORKSHOPS)) {
21         return this.executeGetAllWorkshopsAction();
22     } else if (actionName.equals(GET_WORKSHOP)) {
23         return this.executeGetOneWorkshopAction(parameters);
24     } else {
25         return this.executeDefaultAction();
26     }
27 }
28
29 private String executeCreateWorkshopAction(final Map<String, String> parameters) {
30     int workshopId = workshopCounter++;
31     String workshopName = parameters.get(key:"name");
32     List<String> workshopSections = Arrays.asList(parameters.get(key:"sections").split(regex:","));
33     Workshop workshop = new Workshop(workshopId, workshopName, workshopSections);
34     workshopRepository.add(workshop);
35     return this.workshopSerializer.toJson(workshop);
36 }
37
38 private String executeGetAllWorkshopsAction() {
39     Collection<Workshop> workshops = this.workshopRepository.getAll();
40     return this.workshopSerializer.toJson(workshops);
41 }
42
43 private String executeGetOneWorkshopAction(final Map<String, String> parameters) {
44     String workshopName = parameters.get(key:"name");
45     Workshop workshop = this.workshopRepository.get(workshopName);
46     return this.workshopSerializer.toJson(workshop);
47 }
48
49 private String executeDefaultAction() {
50     return "default";
51 }
```

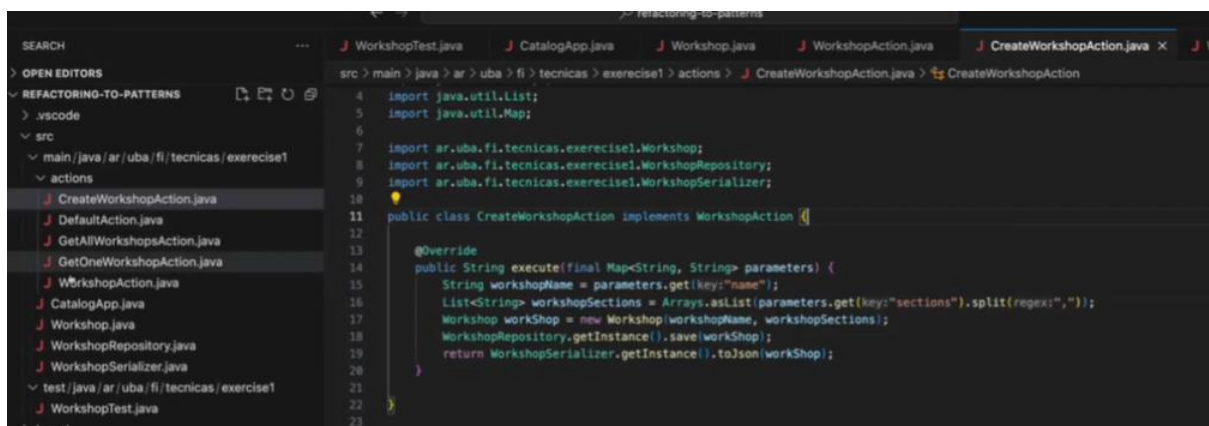
El próximo paso sería que cada función sea una implementación diferente de la clase abstracta strategy, y que cada uno lo implemente de forma diferente. En el caso de la función que no recibe parámetros, se los puedo pasar igual y después no los uso.

También puedo pasarle la responsabilidad de settear al id al workshopRepository, y que ese sea el único que se encargue; en vez de tener que estar pasando el Id en el constructor del Workshop.

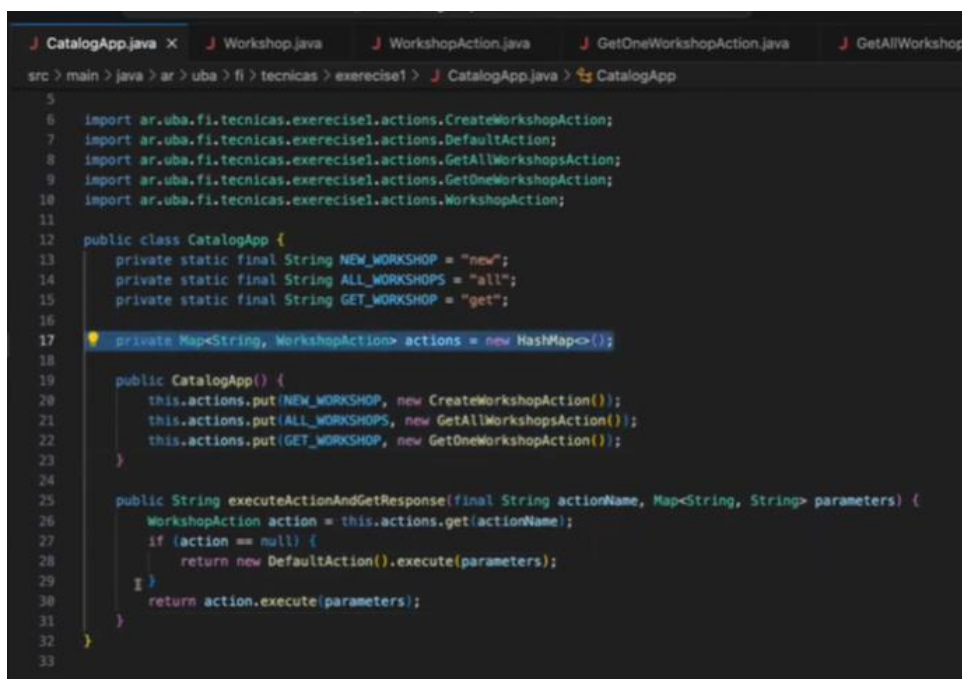
Armo la abstracción:



Esto me permite tener el WorkshopSerializer y WorkshopRepository como “variables globales” en vez de tener que tener el repositorio y el serialzier instanciados en la clase:



Lo que tengo es un conjunto de clases que implementan esta abstracción. Luego, puedo armar un mapa con las acciones posibles y mapear cual tengo que usar en cada caso.



Instancio todas las acciones posibles. Ahora ya no violo OCP. Lo que sí debería tocar si quiero agregar otra, es el mapa, pero ahí ya es un problema de **construcción**. Ahí si yo uso un

framework como spring, podría directamente inyectar las nuevas dependencias en caso de querer agregar algo.

CLASE 3: 17/10 (state, template, adapter)

CASO 1: STATE

Problema

Se desea desarrollar el software para el cajero automático de un banco.

- El cajero comienza inactivo, esperando que se inserte una tarjeta.
- Una vez que el usuario inserta su tarjeta, solicita al usuario que ingrese su PIN.
- Después de que se ingrese un PIN válido, el cajero permite al usuario retirar dinero o verificar su saldo.
- Manejar errores.

Sin usar ningún patrón, una posible solución podría ser (uno de los métodos):

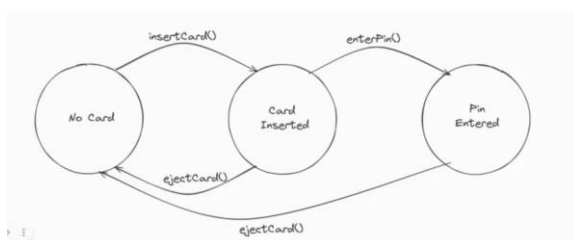
```
public void enterPin(int pin) {  
    if (!cardInserted) {  
        System.out.println("No card inserted.");  
    } else if (pinEntered) {  
        System.out.println("PIN already entered.");  
    } else if (pin == 1234) {  
        pinEntered = true;  
        System.out.println("PIN accepted. You can now perform transactions.");  
    } else {  
        System.out.println("Incorrect PIN. Try again.");  
    }  
}
```

El manejo de errores de los distintos escenarios puede generar muchos condicionales, que le sacan claridad al código, y cada método va a ser de esta forma por el manejo que necesitamos.

Las **acciones** que se pueden realizar o los **errores** que pueden surgir dependen del **estado** del sistema.

Cada vez que quiera agregar un nuevo estado/comportamiento al sistema, estos métodos van a cambiar porque cambia el condicional, tenemos un nuevo caso => es un problema grande

los diferentes estados del problema inicial son:



Son los que hacen que se modifique el comportamiento del sistema.

Una solución se podría dar aplicando el patrón **STATE** (Los estados son quienes tienen que manejar el comportamiento específico).

Se podrían modelar los estados como clases distintas y que cada estado implemente los métodos de manera distinta.

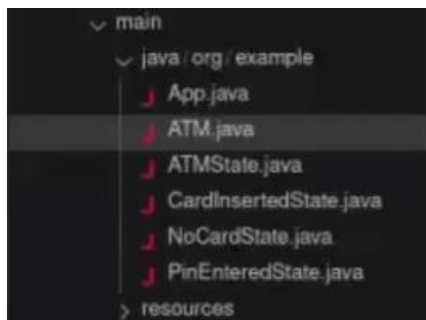
CUANDO USAR STATE: cuando tenemos un sistema cuyo comportamiento cambia dependiendo en el estado en el que este

Código inicial:

```
2
3 public class ATM {
4     private boolean cardInserted;
5     private boolean pinEntered;
6     private double balance;
7
8     public ATM(double balance) {
9         this.cardInserted = false;
10        this.pinEntered = false;
11        this.balance = balance;
12    }
13 }
```

Tengo booleanos para representar los estados y el balance que se le pasa al constructor.

Armo mis estados:



Avance de código

```
public interface ATMState {
    void insertCard(ATM atm);
    void enterPin(ATM atm, int pin);
    void withdraw(ATM atm, double amount);
    void checkBalance(ATM atm);
    void ejectCard(ATM atm);
}
```

Primero hay que definir una interfaz común a todos los estados, los mismos serán los que la implementen.

1. No card state (ej →)
2. Pin entered state
3. Card Inserted state

Esto mismo se hace para cada uno de los estados

OBS: cuando usamos el patrón state, la clase que usa todos los estados se llama **contexto** (en este caso el cajero).

Hago una implementación particular para cada estado.

```
public class NoCardState implements ATMState {
    @Override
    public void insertCard(ATM atm) {
        System.out.println("Card inserted. Please enter your PIN.");
        atm.setState(new CardInsertedState());
    }

    @Override
    public void enterPin(ATM atm, int pin) {
        System.out.println("No card inserted.");
    }

    @Override
    public void withdraw(ATM atm, double amount) {
        System.out.println("No card inserted.");
    }

    @Override
    public void checkBalance(ATM atm) {
        System.out.println("No card inserted.");
    }

    @Override
    public void ejectCard(ATM atm) {
        System.out.println("No card inserted.");
    }
}
```

Avance de código:

¿Una vez que ya tengo implementados todos los estados, como modifico al cajero para que use los mismos y evitar todos los condicionales?

```
public class ATM {
    private double balance;
    private ATMState currentState;

    public ATM(double balance) {
        this.balance = balance;
        this.currentState = new NoCardState();
    }
}
```

Los booleanos no los necesito, solo necesito el estado actual (en este caso ATMState)

```
public void setState(ATMState state) {
    this.currentState = state;
}
```

```
public void insertCard() {
    currentState.insertCard(this);
}

public void enterPin(int pin) {
    currentState.enterPin(this, pin);
}

public void withdraw(double amount) {
    currentState.withdraw(this, amount);
}

public void checkBalance() {
    currentState.checkBalance(this);
}

public void ejectCard() {
    currentState.ejectCard(this);
}
```

Luego cada uno de los métodos que impliquen un cambio de estado, lo único que debe hacer es setear como currentState a sí mismo.

Esto es LUEGO de aplicar el patrón.

IMPORTANTE: los estados deben usar la misma interfaz para que sean reemplazables entre sí, queremos que el comportamiento varíe en runtime.

Logramos sacar todos los condicionales y que los métodos de acá no tengan que modificarse si se agregan nuevos estados.

Mejoras de SOLID a partir de la implementación del patrón:

- **Single responsibility:** le saca todas las responsabilidades al ATM. Cada estado se encarga de lo que corresponde y de nada más.
- **Open close:** podemos agregar más estados al ATM sin tener que tocar el código del mismo, es decir, que los métodos ya implementados no tienen que ser modificados solo vamos a tener que agregar nuevos.

STATE VS STRATEGY

Ambos se enfocan en modificar en runtime, sin embargo tienen algunas diferencias.

Strategy se enfoca en intercambiar diferentes algoritmos o métodos que logran el mismo objetivo, en cambio **State** gestiona diferentes comportamientos según el estado actual del objeto.

En **State** los estados casi siempre conocen a los otros y pueden transicionar entre sí, en **Strategy** esto no pasa casi nunca.

CASO 2: TEMPLATE METHOD (siguiendo con loan de la clase 1)

```
public class TermLoanCapitalStrategy implements CapitalStrategy {  
  
    @Override  
    public double capital(final Loan loan) {  
        return (loan.getCommitment() + loan.getOutstanding()) * loan.getRiskRating();  
    }  
}
```

Tengo la interfaz capital Strategy y dos implementaciones de la misma, TermLoanCapitalStrategy y AdvisedLineCapitalStrategy.

```
public class AdvisedLineCapitalStrategy implements CapitalStrategy {  
  
    @Override  
    public double capital(final Loan loan) {  
        return loan.getCommitment() * loan.getOutstanding() * loan.getRiskRating();  
    }  
}
```

Solo difieren en el operador, uno suma y otro multiplica, es decir que las implementaciones son muy parecidas.

Algo que siempre queremos evitar es la **duplicidad** ya que es una

mala práctica y es difícil de mantener. Cuando hay que hacer un cambio hay que tocar más de un lugar del código.

Hay que encapsular lo que varía, en este caso la suma o el producto. Esto puedo hacerlo en un método abstracto que se encuentre en algún lugar común a los dos, y luego lo implemento diferente en cada caso.

Avance de código:

```
@Override  
public double capital(final Loan loan) {  
    return this.riskAmountFor(loan) * loan.getRiskRating();  
}
```

```
private double riskAmountFor(final Loan loan) {  
    return loan.getCommitment() + loan.getOutstanding();  
}
```

Ahora ambas implementaciones de la interfaz tienen exactamente el mismo método capital.

Eso es lo puedo llevarlo a un lugar común, como por ejemplo a una clase abstracta.

```
private double riskAmountFor(final Loan loan) {  
    return loan.getCommitment() + loan.getOutstanding();  
}
```

Avance de código:

```

public abstract class CapitalStrategy {
    public double capital(final Loan loan) {
        return this.riskAmountFor(loan) * loan.getRiskRating();
    }

    protected abstract double riskAmountFor(final Loan loan);
}

```

Ahora quien extienda (ya no es interfaz sino que clase abstracta) debe decidir cómo se implementa el riskAmountFor, es lo único que cambia entre las estrategias. =>

```

public class TermLoanCapitalStrategy extends CapitalStrategy {
    @Override
    protected double riskAmountFor(final Loan loan) {
        return loan.getCommitment() + loan.getOutstanding();
    }
}

```

```

public class AdvisedLineCapitalStrategy extends CapitalStrategy {
    @Override
    protected double riskAmountFor(final Loan loan) {
        return loan.getCommitment() * loan.getOutstanding();
    }
}

```

Aca estamos evitando duplicidad usando herencia.

Usando composicion tambien se podria haciendo algo como lo siguiente:
strategy de strategy

```

public abstract class CapitalStrategy {
    private Algo algo;

    public double capital(final Loan loan) {
        return algo.riskAmountFor(loan) * loan.getRiskRating();
    }
}

```

Esta idea de hacer un método dejarlo abstracto para que lo implementen sus derivadas se conoce como **TEMPLATE METHOD**. Se define una plantilla que hay que completar y ahora el algoritmo (el de capital en este ejemplo) está encapsulado.

CASO 3: ADDAPTER

Tengo una clase test llamada AudioPlayerTest

```

public class AudioPlayerTest {
    @Test
    public void testPlayMusic() {
        AudioPlayer audioPlayer = new AudioPlayer();

        audioPlayer.playMusic(audioType:"mp3", fileName:"song1.mp3");
        audioPlayer.playMusic(audioType:"mp4", fileName:"song2.mp4");
        audioPlayer.playMusic(audioType:"vlc", fileName:"song3.vlc");
        audioPlayer.playMusic(audioType:"xyz", fileName:"song4.avi");
    }
}

```

Tambien tengo una clase MediaPlayer y AudioPlayer (implementa MediaPlayer)

```

public interface MediaPlayer {
    public void playMusic(String audioType, String fileName);
}

```

```

public class AudioPlayer implements MediaPlayer {
    @Override
    public void playMusic(String audioType, String fileName) {
        if (audioType.equalsIgnoreCase("mp3")) {
            System.out.println("Playing mp3 file: " + fileName);
        } else {
            System.out.println("The given format: " + audioType + " is not supported");
        }
    }
}

```

por otro lado tengo un paquete llamado `advancedMediaPlayer` que tiene:

```
public interface AdvancedMediaPlayer {  
    public void playVlcPlayer(String fileName);  
    public void playMp4Player(String fileName);  
}
```

Tengo las implementaciones de `MP4AdvancedMediaPlayer` y `VLCAdvancedMediaPlayer`. Vemos que cada uno solo implementa el método de la interfaz que le interesa, y el otro lo deja el blanco. A una no le importa la otra, solo implementa lo que necesita y el resto lo deja

```
public class VLCAdvancedMediaPlayer implements AdvancedMediaPlayer {  
    @Override  
    public void playVlcPlayer(String fileName) {  
        System.out.println("Playing vlc file: " + fileName);  
    }  
  
    @Override  
    public void playMp4Player(String fileName) {  
        //do nothing  
    }  
}
```

```
public class MP4AdvancedMediaPlayer implements AdvancedMediaPlayer {  
    @Override  
    public void playVlcPlayer(String fileName) {  
        //do nothing  
    }  
  
    @Override  
    public void playMp4Player(String fileName) {  
        System.out.println("Playing mp4 file: " + fileName);  
    }  
}
```

como "do nothing".

Esto viola el principio de **segregación de interfaces**.

A mi me gustaría que mi `AudioPlayer` soporte VLC y MP4, ahora solo soporta MP3. ¿Cómo hago eso? (el paquete `advancedMediaPlayer` no lo puedo tocar, como si fuese una librería)

Avance de código:

Agrego al `if` de `AudioPlayer` los casos de VLC y MP4. Dentro del mismo delego a cada uno según corresponda.

```
public class AudioPlayer implements MediaPlayer {  
    private MP4AdvancedMediaPlayer mp4AdvancedMediaPlayer = new MP4AdvancedMediaPlayer();  
    private VLCAdvancedMediaPlayer vlcAdvancedMediaPlayer = new VLCAdvancedMediaPlayer();  
  
    @Override  
    public void playMusic(String audioType, String fileName) {  
        if (audioType.equalsIgnoreCase("mp3")) {  
            System.out.println("Playing mp3 file: " + fileName);  
        } else if (audioType.equalsIgnoreCase("mp4")) {  
            this.mp4AdvancedMediaPlayer.playMp4Player(fileName);  
        } else if (audioType.equalsIgnoreCase("vlc")) {  
            this.vlcAdvancedMediaPlayer.playVlcPlayer(fileName);  
        } else {  
            System.out.println("The given format: " + audioType + " is not supported");  
        }  
    }  
}
```

Ahora me gustaría que sea extensible a cualquier tipo de audio sin necesidad de volver a agregar ifs.

Esto lo puedo lograr con polimorfismo.

Como lo de `advanced` no lo puedo tocar => creo una clase que implemente la abstracción

y que wrappee a una de las implementadas en el paquete.

Avance de código:

Creo la abstracción `SimpleMediaPlayer`:

```
public interface SimpleMediaPlayer {
    public void play(String fileName);
}
```

También tengo implementaciones como:

```
public class MP3SimpleMediaPlayer implements SimpleMediaPlayer {
    @Override
    public void play(String fileName) {
        System.out.println("Playing mp3 file: " + fileName);
    }
}
```

```
public class MP4SimpleMediaPlayer implements SimpleMediaPlayer {
    private MP4AdvancedMediaPlayer mp4AdvancedMediaPlayer = new MP4AdvancedMediaPlayer();

    @Override
    public void play(String fileName) {
        this.mp4AdvancedMediaPlayer.playMp4Player(fileName);
    }
}
```

En el caso de MP4 implemento la abstracción y en el método play delego a una instancia de las que no puedo tocar.

```
public class AudioPlayer implements MediaPlayer {
    private SimpleMediaPlayer mp3SimpleMediaPlayer = new MP3SimpleMediaPlayer();
    private SimpleMediaPlayer mp4SimpleMediaPlayer = new MP4SimpleMediaPlayer();
    private SimpleMediaPlayer vlcSimpleMediaPlayer = new VLCSimpleMediaPlayer();

    @Override
    public void playMusic(String audioType, String fileName) {
        if (audioType.equalsIgnoreCase("mp3")) {
            this.mp3SimpleMediaPlayer.play(fileName);
        } else if (audioType.equalsIgnoreCase("mp4")) {
            this.mp4SimpleMediaPlayer.play(fileName);
        } else if (audioType.equalsIgnoreCase("vlc")) {
            this.vlcSimpleMediaPlayer.play(fileName);
        } else {
            System.out.println("The given format: " + audioType + " is not supported");
        }
    }
}
```

Ahora el AudioPlayer en todos los casos hace lo mismo (hace play):

A continuación mi objetivo es eliminar los if else. Lo que quiero es configurar por inyección cuales son los mediaPlayer concretos. Una de las formas es tener un mapa.

Avance de código:

```
public class AudioPlayer implements MediaPlayer {
    private Map<String, SimpleMediaPlayer> mediaPlayers = new HashMap<>();

    public void addMediaPlayer(String audioType, final SimpleMediaPlayer mediaPlayer) {
        this.mediaPlayers.put(audioType, mediaPlayer);
    }

    @Override
    public void playMusic(String audioType, String fileName) {
        SimpleMediaPlayer mediaPlayer = this.mediaPlayers.get(audioType);
        if (mediaPlayer == null) {
            System.out.println("The given format: " + audioType + " is not supported");
        } else {
            mediaPlayer.play(fileName);
        }
    }
}
```

Al hacer play entro al mapa y busco el tipo de audio, si lo encuentro hago play y sino error.


```

public class AudioPlayerTest {
    @Test
    public void testPlayMusic() {
        final SimpleMediaPlayer mp3SimpleMediaPlayer = new MP3SimpleMediaPlayer();
        final SimpleMediaPlayer mp4SimpleMediaPlayer = new MP4SimpleMediaPlayer();
        final SimpleMediaPlayer vlcSimpleMediaPlayer = new VLCSimpleMediaPlayer();

        AudioPlayer audioPlayer = new AudioPlayer();
        audioPlayer.addMediaPlayer(audioType:"mp3", mp3SimpleMediaPlayer);
        audioPlayer.addMediaPlayer(audioType:"mp4", mp4SimpleMediaPlayer);
        audioPlayer.addMediaPlayer(audioType:"vlc", vlcSimpleMediaPlayer);

        audioPlayer.playMusic(audioType:"mp3", fileName:"song1.mp3");
        audioPlayer.playMusic(audioType:"mp4", fileName:"song2.mp4");
        audioPlayer.playMusic(audioType:"vlc", fileName:"song3.vlc");
        audioPlayer.playMusic(audioType:"xyz", fileName:"song4.avi");
    }
}

```

Ahora todas las construcciones están en el test, también la configuración y ejecución del play. El tema de construcción podría resolverlo con un factory por ejemplo y así sacarlo del test, de esta manera encapsulo el código de creación. Otra opción podría ser mover eso al main del sistema. (pseudocódigo)

```

public class AudioPlayerFactory {
    public AudioPlayer createAudioPlayer(Map<String, SimpleMediaPlayer> mediaPlayers) {
        AudioPlayer audioPlayer = new AudioPlayer();

        for (entry in mediaPlayers) {
            String type = entry[0];
            SimpleMediaPlayer mediaPlayer = entry[1];
            audioPlayer.addMediaPlayer(type, mediaPlayer);
        }

        return audioPlayer;
    }
}

```

La idea de wrappear una clase para que cumpla con una interfaz (relación de composición) se conoce con el nombre de **ADAPTER**.

CUANDO USAR ADAPTER: Este patrón es útil cuando tengo una clase que no conforma con la abstracción que quiero entonces la compongo. Evito la segregación de interfaces.

CLASE 4: 24/10 (intérprete, chain of responsibility, null object)

CASO 1: INTERPRETER

```

public class ProductFinder {
    private final ProductRepository repository;

    public ProductFinder(final ProductRepository repository) {
        this.repository = repository;
    }

    public Collection<Product> byColor(Color color) {
        Collection<Product> foundProducts = new ArrayList<>();
        Collection<Product> allProducts = this.repository.getAll();
        allProducts.forEach(product -> {
            if (product.getColor() == color) {
                foundProducts.add(product);
            }
        });
        return foundProducts;
    }

    public Collection<Product> byPrice(double price) {
        Collection<Product> foundProducts = new ArrayList<>();
        Collection<Product> allProducts = this.repository.getAll();
        allProducts.forEach(product -> {
            if (product.getPrice() == price) {
                foundProducts.add(product);
            }
        });
        return foundProducts;
    }
}

```

Quiero agregar distintos tipos de búsqueda.

Intento hacer un método “findBy” en el que le paso el valor por el cual buscar y el criterio o condición. Hay que crear una abstracción nueva que encapsule la condición del if.

Podemos crear una función que sea el parámetro de condición.

Avance de código:

```

@Test
public void testFindByColorSizeAndBelowPrice() {
    Collection<Product> foundProducts = this.finder.byColorSizeAndBelowPrice(Color.red, ProductSize.SMALL, price:10.00);
    assertEquals(expected:0, foundProducts.size());

    foundProducts = this.finder.byColorSizeAndBelowPrice(Color.red, ProductSize.MEDIUM, price:10.00);
    assertEquals(expected:1, foundProducts.size());
    assertTrue(foundProducts.contains(fireTruck));
}

```

Le pasa por parametro todo lo que necesita, el color, el tamaño y price que actuaría como cota superior.

```

public Collection<Product> byColorSizeAndBelowPrice(Color color, ProductSize size, double price) {
    Collection<Product> foundProducts = new ArrayList<>();
    Collection<Product> allProducts = this.repository.getAll();
    allProducts.forEach(product -> {
        if (product.getColor() == color && product.getSize() == size && product.getPrice() < price) {
            foundProducts.add(product);
        }
    });
    return foundProducts;
}

```

El método hace la iteración y el if por los parámetros dados.

Todos los métodos que impliquen un nuevo método de búsqueda tendrán la misma lógica => el objetivo es encapsular los ifs y no tener muchos métodos en los que lo único que cambia es el condicional.

Avance de código:

```

public Collection<Product> findBy(final ProductSpec spec) {
    Collection<Product> foundProducts = new ArrayList<>();
    Collection<Product> allProducts = this.repository.getAll();
    allProducts.forEach(product -> {
        if (spec.isSatisfiedBy(product)) {
            foundProducts.add(product);
        }
    });
    return foundProducts;
}

```

La abstracción en este caso sería ProductSpec, la misma cumple con ser un método que evalúa si el producto cumple con la especificación o no y además contiene los valores.

```

public interface ProductSpec {
    public boolean isSatisfiedBy(final Product product);
}

```

La abstracción tiene un método booleano que indica si la condición es satisfecha o no.

```

public class ColorSpec implements ProductSpec {
    private Color color;

    public ColorSpec(final Color color) {
        this.color = color;
    }

    @Override
    public boolean isSatisfiedBy(Product product) {
        return product.getColor() == this.color;
    }
}

```

Por ejemplo si queremos buscar por color, ColorSpec recibe como parámetro el color y el método isSatisfiedBy toma el producto y evalúa contra el color condicional y devuelve el booleano correspondiente.

Ahora el test crea una especificación con el color de condición que nos importa y luego hago un findBy con esa especificación.

Avance de código:

```
@Test
void ar.uba.fi.tecnicas.exercisel.ProductFinderTest.te
public void testFindByColor() {
    ColorSpec spec = new ColorSpec(Color.red);
    Collection<Product> foundProducts = this.finder.findBy(spec);
    assertEquals(expected:2, foundProducts.size());
    assertTrue(foundProducts.contains(fireTruck));
    assertTrue(foundProducts.contains(toyConvertible));
}
```

Ahora quiero aplicar la búsqueda a price.

Creo la clase PriceSpec que implementa la interfaz ProductSpec.

Esa es justamente la idea del diseño, generar una abstracción que encapsule todo lo que varía.

```
public class PriceSpec implements ProductSpec {
    private double price;

    public PriceSpec(final double price) {
        this.price = price;
    }

    @Override
    public boolean isSatisfiedBy(final Product product) {
        return product.getPrice() == this.price;
    }
}
```

Ahora el test:

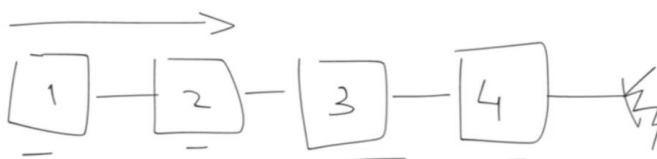
```
@Test
public void testFindByPrice() {
    PriceSpec spec = new PriceSpec(price:8.95);
    Collection<Product> foundProducts = this.finder.findBy(spec);
    assertEquals(expected:2, foundProducts.size());
    foundProducts.forEach(product -> {
        assertTrue(product.getPrice() == 8.95);
    });
}
```

Ahora... ¿qué pasa cuando quiero combinar?

Puedo hacer una nueva spec con la combinación sin embargo pueden haber combinaciones y criterios infinitos y siempre tendría que crear una nueva clase. ¿Cómo hago para que no explote?

EXTRA:

Chain of responsibility



Tengo una secuencia de pasos y mando a ejecutar la siguiente. En una cadena, cada eslabón tiene la misma interfaz con algún método como por ejemplo evaluar.

La idea del patrón es construir una cadena la cual recibe algo a

evaluar y alguno de los eslabones se hace cargo de llevarlo a cabo, no me interesa quien lo

hace. Cuando un eslabón logra evaluar la condición, corta la cadena; si no puede se lo pasa al siguiente. Los eslabones son siempre los mismos.

En este caso no podría aplicarse ya que necesito que todos evalúen y si todos los eslabones devuelve true entonces es true toda la validación, este no es el foco del parton.

```
public class AndProductSpec implements ProductSpec {
    private final ProductSpec left;
    private final ProductSpec right;

    public AndProductSpec(final ProductSpec left, final ProductSpec right) {
        this.left = left;
        this.right = right;
    }

    @Override
    public boolean isSatisfiedBy(final Product product) {
        return this.left.isSatisfiedBy(product) && this.right.isSatisfiedBy(product);
    }
}
```

Nos gustaría crear una estructura dinámica pensada como condiciones unidas por operadores. Ahora encapsulo el operador (AND en este caso). Implementó la misma interfaz y le paso dos specs si ambas se satisfacen => se satisface todo.

```
public class BelowPriceSpec implements ProductSpec {
    private final double price;

    public BelowPriceSpec(final double price) {
        this.price = price;
    }

    @Override
    public boolean isSatisfiedBy(Product product) {
        return product.getPrice() < this.price;
    }
}
```

También se implementa la "below price".

Ahora el test hago un and entre el color que queria (rojo) y el size buscado (small) y valido:

```
@Test
public void testFindByColorSizeAndBelowPrice() {
    ProductSpec spec = new AndProductSpec(new AndProductSpec(new ColorSpec(Color.red), new SizeSpec(ProductSize.SMALL)), new BelowPriceSpec(100));
    Collection<Product> foundProducts = this.finder.findBy(spec);
    assertEquals(expected:0, foundProducts.size());

    spec = new AndProductSpec(new AndProductSpec(new ColorSpec(Color.red), new SizeSpec(ProductSize.MEDIUM)), new BelowPriceSpec(100));
    foundProducts = this.finder.findBy(spec);
    assertEquals(expected:1, foundProducts.size());
    assertTrue(foundProducts.contains(fireTruck));
}
```

Ahora todos los métodos del productFinder no tienen sentido, solo necesito el findBy, y todas las clases que implementan los operadores.

```

src > main > java > ar > uba > fi > tecnicas > ejercicio1 > ProductFinder.java > ...
1  package ar.uba.fi.tecnicas.ejercicio1;
2
3  import java.util.ArrayList;
4  import java.util.Collection;
5
6  import ar.uba.fi.tecnicas.ejercicio1.spec.ProductSpec;
7
8  public class ProductFinder {
9      private final ProductRepository repository;
10
11     public ProductFinder(final ProductRepository repository) {
12         this.repository = repository;
13     }
14
15     public Collection<Product> findBy(final ProductSpec spec) {
16         Collection<Product> foundProducts = new ArrayList<>();
17         Collection<Product> allProducts = this.repository.getAll();
18         allProducts.forEach(product -> {
19             if (spec.isSatisfiedBy(product)) {
20                 foundProducts.add(product);
21             }
22         });
23         return foundProducts;
24     }
25
26 }
27

```

Una opción de patrón para encapsular la Query puede ser BUILDER ya que el objetivo del código es tener una abstracción que encapsule la condición y luego tener implementaciones simples y más complejas, permitiendo queries tan complejas como quiera. El objetivo de builder es ir construyendo la especificación. Puedo usarlo para que la query no quede tan choclo y quede más simple cuando las consultas son complejas.

```

ProductSpecBuilder builder = new ProductSpecBuilder(...);
builder.and(builder.and(builder.withColor(Color.red), builder.withSize(ProductSize.SMALL)))

```

pseudocódigo de idea.

La idea de encapsular una query con composición de objetos y que cada uno de ellos representa la abstracción, recibe el nombre de **INTÉRPRETE**.

CASO 2: NULL OBJECT

```

public class MouseEventHandler {

    public boolean mouseMove(int x, int y) {
        // DO SOMETHING ...
        return false;
    }

    public boolean mouseDown(int x, int y) {
        // DO SOMETHING ...
        return false;
    }

    public boolean mouseUp(int x, int y) {
        // DO SOMETHING ...
        return false;
    }

    public boolean mouseExit(int x, int y) {
        // DO SOMETHING ...
        return false;
    }

}

```

```

public class NavigationApplet {
    private MouseEventHandler mouseEventHandler;

    public void setMouseEventHandler(MouseEventHandler mouseEventHandler) {
        this.mouseEventHandler = mouseEventHandler;
    }

    public boolean mouseMove(int x, int y) {
        if (this.mouseEventHandler != null) {
            return this.mouseEventHandler.mouseMove(x, y);
        } else {
            return true;
        }
    }

    public boolean mouseDown(int x, int y) {
        if (this.mouseEventHandler != null) {
            return this.mouseEventHandler.mouseDown(x, y);
        } else {
            return true;
        }
    }

    public boolean mouseUp(int x, int y) {
        if (this.mouseEventHandler != null) {
            return this.mouseEventHandler.mouseUp(x, y);
        } else {
            return true;
        }
    }

    public boolean mouseExit(int x, int y) {
        if (this.mouseEventHandler != null) {
            return this.mouseEventHandler.mouseExit(x, y);
        } else {
            return true;
        }
    }
}

```


En todos los métodos del NavigationApplet hago lo mismo: chequeo si el atributo es NULL y en caso de que lo sea siempre hago lo mismo, vuelvo a tener el problema de duplicidad. Para la parte del else, podría llamar a un NullHandler que se encargue de devolver lo que corresponde.

```
public class NullMouseEventHandler extends MouseEventHandler {  
    public boolean mouseMove(int x, int y) {  
        return true;  
    }  
  
    public boolean mouseDown(int x, int y) {  
        return true;  
    }  
  
    public boolean mouseUp(int x, int y) {  
        return true;  
    }  
  
    public boolean mouseExit(int x, int y) {  
        return true;  
    }  
}
```

Inicialmente seteo el atributo en null. Si nunca llamo al setter devuelvo el atributo que va a tener

```
public class NavigationApplet {  
    private MouseEventHandler mouseEventHandler = new NullMouseEventHandler();  
  
    public void setMouseEventHandler(MouseEventHandler mouseEventHandler) {  
        if (mouseEventHandler == null) {  
            throw new RuntimeException(message:"Mouse event handler cannot be null");  
        }  
        this.mouseEventHandler = mouseEventHandler;  
    }  
  
    public boolean mouseMove(int x, int y) {  
        return this.mouseEventHandler.mouseMove(x, y);  
    }  
  
    public boolean mouseDown(int x, int y) {  
        return this.mouseEventHandler.mouseDown(x, y);  
    }  
  
    public boolean mouseUp(int x, int y) {  
        return this.mouseEventHandler.mouseUp(x, y);  
    }  
  
    public boolean mouseExit(int x, int y) {  
        return this.mouseEventHandler.mouseExit(x, y);  
    }  
}
```

al null entonces devolverá true. Va a seguir siendo lo mismo. Ya no tengo que manejar la nulidad, nunca es null en realidad.

La idea de crear un objeto que representa la nulidad se conoce con el nombre **NULL OBJECT**. El mismo representa qué hacer en caso de null, no todos los casos tienen que hacer lo mismo sino que puedo hacer una implementación (handler) diferente para cada caso.

CLASE 5: 28/10 (visitor)

CASO 1: VISITOR

Problema:

Se tiene un sistema con productos de diferentes tipos. Ropa, electrónica y alimentos.

- Se desea implementar nuevos comportamientos
 - Poder calcular el descuento para cada tipo de producto según sus características.
 - Poder exportar los productos a JSON.
- **Requisito especial:** Las clases de los productos deben cambiar lo menos posible!

Tengo una interfaz producto con dos métodos a implementar: getName y getDiscount.

```
public class App {  
    Run | Debug  
    public static void main(String[] args) {  
        List<Product> products = new ArrayList<>();  
  
        products.add(new GroceryProduct(name:"Orange", LocalDate.now().plusDays(1), weightInKg:1.0, priceByKg:10));  
        products.add(new ElectronicsProduct(brand:"Samsung", model:"Galaxy S21", price:999.99));  
        products.add(new ClothingProduct(name:"T-Shirt", size:"M", price:19.99));  
  
        for (Product product : products) {  
            System.out.println("Name: " + product.getName());  
            System.out.println("Discount: $" + String.format("%.2f", product.getDiscount()));  
        }  
    }  
}
```

Un primer acercamiento al conflicto de los descuentos es que cada producto sepa cómo calcular su propio descuento.

La idea principal es no modificar tanto cada una de las clases que implementan la interfaz.

Se intentará mejorar ese aspecto usando **VISITOR**. La idea de este patrón es que sepa 'visitar' a cada elemento y aplicar, en este caso, el descuento que corresponda.

Avance de código:

```
public class ElectronicsProduct implements Product {  
    private String brand;  
    private String model;  
    private double price;  
  
    public ElectronicsProduct(String brand, String model, double price) {  
        this.brand = brand;  
        this.model = model;  
        this.price = price;  
    }  
  
    public String getBrand() {  
        return brand;  
    }  
  
    public String getModel() {  
        return model;  
    }  
  
    public String getName() {  
        return brand + " " + model;  
    }  
  
    public double getPrice() {  
        return price;  
    }  
}
```

Cada uno de los productos concretos ya no implementa los descuentos y la interfaz producto tampoco tiene ese método.

```

public class DiscountVisitor {
    public double visit(GroceryProduct product) {
        if (product.getExpiryDate().isBefore(LocalDate.now().plusDays(7))) {
            return product.getPrice() * 0.1;
        }
        return 0;
    }

    public double visit(ElectronicsProduct product) {
        if (product.getBrand().equals("Samsung")) {
            return product.getPrice() * 0.3;
        }
        return 0;
    }

    public double visit(ClothingProduct product) {
        if (product.getSize().equals("M")) {
            return product.getPrice() * 0.1;
        }
        return 0;
    }
}

```

Esta clase sabe visitar a cada uno de los productos y aplicar los descuentos según corresponda.

Java permite crear varios métodos con el mismo nombre, pero no todos lo hacen.

Ahora en el main:

```

for (Product product : products) {
    System.out.println("Name: " + product.getName());
    System.out.println("Discount: $" + String.format("%.2f", discountVisitor.visit(product)));
}

```

El error que se marca en

el `.visit()` tiene que ver con que no hay suficiente información para poder determinar el tipo del producto concreto.

Avance de código:

```

public class App {
    public static void main(String[] args) {
        List<Product> products = new ArrayList<>();

        products.add(new GroceryProduct(name:"Orange", LocalDate.now().plusDays(1), weightInKg:1.0, priceByKg:10));
        products.add(new ElectronicsProduct(brand:"Samsung", model:"Galaxy S21", price:999.99));
        products.add(new ClothingProduct(name:"T-Shirt", size:"M", price:19.99));

        DiscountVisitor discountVisitor = new DiscountVisitor();

        for (Product product : products) {
            System.out.println("Name: " + product.getName());
            System.out.println("Discount: $" + String.format("%.2f", product.accept(discountVisitor)));
        }
    }
}

```

```

public interface Product {
    String getName();

    double accept(DiscountVisitor visitor);
}

```

Ahora la interfaz tiene el método `accept` que cada uno de los productos concretos va a tener que implementar.

En este momento this no tiene el conflicto de qué tipo concreto tiene porque está dentro de la misma clase.

```

@Override
public double accept(DiscountVisitor visitor) {
    return visitor.visit(this);
}

```

Avance de código:

```
public class ExportToJSONVisitor {
    public String visit(GroceryProduct product) {
        return String.format(
            "{\\\"type\\\":\\\"GroceryProduct\\\",\\\"name\\\":\\\"%s\\\",\\\"price\\\":%.2f,\\\"expirationDate\\\":\\\"%s\\\"}",
            product.getName(),
            product.getPrice(),
            product.getExpiryDate());
    }

    public String visit(ElectronicsProduct product) {
        return String.format(
            "{\\\"type\\\":\\\"ElectronicsProduct\\\",\\\"name\\\":\\\"%s\\\",\\\"price\\\":%.2f,\\\"brand\\\":\\\"%s\\\",\\\"model\\\":\\\"%s\\\"}",
            product.getName(),
            product.getPrice(),
            product.getBrand(),
            product.getModel());
    }

    public String visit(ClothingProduct product) {
        return String.format(
            "{\\\"type\\\":\\\"ClothingProduct\\\",\\\"name\\\":\\\"%s\\\",\\\"price\\\":%.2f,\\\"size\\\":\\\"%s\\\"}",
            product.getName(),
            product.getPrice(),
            product.getSize());
    }
}
```

Ahora... implementamos el JSON para darle más sentido al uso de visitor.

Sin embargo, si no hago una interfaz general de visitor, ahora cada producto debería crear otro accept para otro tipo de visitar y eso agregaría, más complejidad.

```
public interface Visitor {
    void visit(GroceryProduct product);

    void visit(ElectronicsProduct product);

    void visit(ClothingProduct product);
}
```

```
@Override
public void accept(Visitor visitor) {
    visitor.visit(this);
}
```

Ahora el accept toma un visitor general, no un discountVisitor.

```
public class ExportToJSONVisitor implements Visitor {
    // ...

    for (Product product : products) {
        System.out.println("Name: " + product.getName());
        product.accept(discountVisitor);
        product.accept(exportToJSONVisitor);
    }
}
```

Ventajas:

- **Single Responsibility Principle:** Separa los comportamientos de los objetos.
- **Open/Closed Principle:** Permite agregar nuevas operaciones fácilmente, sin modificar las clases de los objetos visitados.

A visitor object can **accumulate** some useful information while working with various objects. This might be handy when you want to traverse some complex object structure, such as an object tree, and apply the visitor to each object of this structure.

Un visitor lo que puede hacer es ir guardando la información que puede ser útil. Por ejemplo, usamos un visitor para calcular el tamaño de los archivos en un file system, iteramos sobre la

estructura lo cual puede ser complejo, pero le damos al visitor la potestad de poder guardarse los datos y después calcular cosas como el peso total del sistema.

Desventajas:

- Romper encapsulamiento
- Alto acoplamiento con las clases visitadas
- Agregar un nuevo tipo de objeto -> Modificar todos los visitor

Rompe el encapsulamiento porque el mismo está altamente ligado con las clases que visita, necesita saber muchas cosas de ellas (como por ejemplo el precio).

CUANDO USAR VISITOR: este patrón conviene usarlo cuando lo que varía (o lo que quiero agregar) son formas de atravesar o extraer información de una jerarquía, como por ejemplo exportar, calcular un valor. Se puede aplicar cuando tenemos una estructura, como un árbol, y queremos acceder a su información.

Si lo que varía es la jerarquía (ej: agregar un nuevo tipo de producto) hay que tocar todos los visitors, eso no es conveniente. Lo ideal es usarlo cuando se que la estructura es bastante sólida y no va a cambiar mucho.