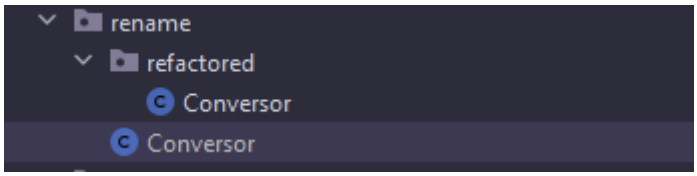
2023

# REFACTOTIZACIÓN

HITO 5.4

MIJAEL TAMAYO ONOFRE

# RENAME




CÓDIGO SIN REFACTORIZAR

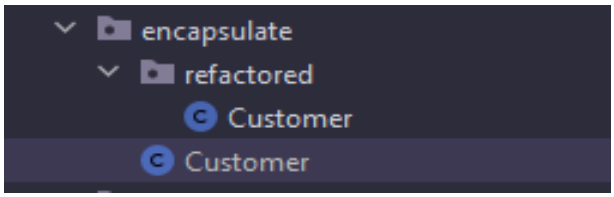
```
public class Conversor {
    2 usages
    public float conv (float c) {
        float x = c * 166.386f;
        return x;
    }
}
```

CÓDIGO REFACTORIZADO

Este método lo que hace es convertir euros a pesetas. En este código observamos que el valor “166.386f” se guarda dentro de una variable de tipo static final y luego es usado dentro de la función. Se crea dentro de un variable para poder usarlo dentro de otros métodos.

```
public class Conversor {
    1 usage
    private static final float EUROS_PESETAS_CHANGE_RATE = 166.386f;

    no usages
    public float eurosToPesetas (float euros) {
        float pesetas = euros * EUROS_PESETAS_CHANGE_RATE;
        return pesetas;
    }
}
```

# ENCAPSULATE




CÓDIGO SIN REFACTORIZAR

```java
public class Customer {

    2 usages
    String name;
    2 usages
    int id;

    public Customer() {
        init();
    }

    1 usage
    public void init() {
        name = "Eugene Krabs";
        id = 42;
    }

    public String toString() {
        return id + ":" + name;
    }
}
```

Este código sin refactorizar tiene variables públicas y se puede acceder desde otras clases.

El método init() lo que hace es inicializar los campos de la clase Customer.

El método toString() utiliza directamente los campos "name" e "id" en lugar de utilizar métodos de acceso.

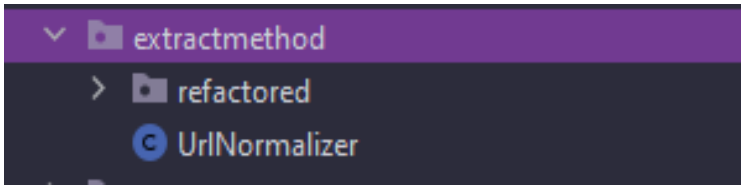
CÓDIGO REFACTORIZADO

```java
public class Customer {

    2 usages
    private String name;
    2 usages
    private int id;

    public Customer() {
        init();
    }

    1 usage
    private void init() {
        setName("Eugene Krabs");
        setId(42);
    }

    public String toString() {
        return getId() + ":" + getName();
    }
```

En el código refactorizado observamos que name e id ahora son privadas. También tiene getters and setters y el método init() es privado.

El método toString() utiliza los setters del name e id del Customer.

# EXTRACTMETHOD




CÓDIGO SIN REFACTORIZAR

```java
public class UrlNormalizer {

    1 usage
    public String normalize(String title) {
        String url = "";
        // First we trim whitespaces
        url = title.trim();

        // Remove special chars
        String specialRemoved = "";
        for (int i = 0; i < url.length(); i++) {
            if (url.charAt(i) != ',' && url.charAt(i) != ':'
                    && url.charAt(i) != '.' && url.charAt(i) != '?') {
                specialRemoved += url.charAt(i);
            }
        }

        url = specialRemoved;

        // Replace white spaces with hyphens
        String spacesReplaced = "";
        for (int i = 0; i < url.length(); i++) {
            if (url.charAt(i) == ' ') {
                spacesReplaced += "-";
            } else {
                spacesReplaced += url.charAt(i);
            }
        }
        url = spacesReplaced;

        // lowercase everything
        url = url.toLowerCase();

        return url;
    }
}
```

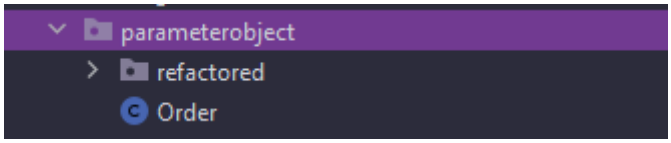
Este código nos muestra que el método normalize() normaliza una cadena de texto. También podemos observar que todo está dentro del mismo método.

CÓDIGO REFACTORIZADO

```
no usages
public String normalize(String title) {
    String url = trimSpaces(title);

    url = removeSpecialChars(url);
    url = replaceSpaces(url);
    url = url.toLowerCase();

    return url;
}

1 usage
private String replaceSpaces(String url) {
    String spacesReplaced = "";
    for (int i = 0; i < url.length(); i++) {
        if (url.charAt(i) == ' ') {
            spacesReplaced += "-";
        } else {
            spacesReplaced += url.charAt(i);
        }
    }
    url = spacesReplaced;
    return url;
}

1 usage
private String removeSpecialChars(String url) {
    String specialRemoved = "";
    for (int i = 0; i < url.length(); i++) {
        if (url.charAt(i) != ',' && url.charAt(i) != ':'
                && url.charAt(i) != '.' && url.charAt(i) != '?') {
            specialRemoved += url.charAt(i);
        }
    }
    url = specialRemoved;
    return url;
}
    1 usage
    private String trimSpaces(String title) {
        String url = "";
        url = title.trim();
        return url;
    }
}
```

Este código nos muestra que el método normalize() se han agregado tres métodos nuevos replaceSpaces() removeSpecialChars() trimSpaces(). El método normalize() ahora llama a estos tres métodos.

# PARAMETEROBJECT




CÓDIGO SIN REFACTORIZAR

El método addItem() recibe como parámetros muchas variables.

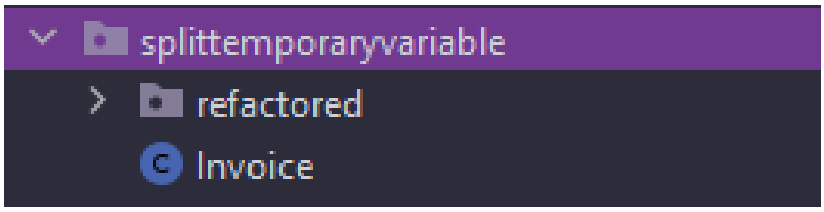
```
public class Order {
    3 usages
    private Hashtable<String, Float> items = new Hashtable<String, Float>();

    3 usages
    public void addItem(Integer productID, String description, Integer quantity, Float price, Float discount) {
        items.put(productID + ": " + description, (quantity * price) - (quantity * price * discount));
    }

    1 usage
    public float calculateTotal() {
        float total = 0;
        Enumeration<String> keys = items.keys();

        while (keys.hasMoreElements()) {
            total = total + items.get(keys.nextElement());
        }
        return total;
    }
}
```

CÓDIGO REFACTORIZADO

El método addItem() recibe un Objeto en lugar de muchas variables . Dentro de la clase OrderItem tiene getters que llama a sus variables.

```
public class Order {
    3 usages
    private Hashtable<String, Float> items = new Hashtable<String, Float>();

    no usages
    public void addItem (OrderItem orderItem) {
        items.put(orderItem.getProductID() + ": " + orderItem.getDescription(), orderItem.totalItem());
    }

    no usages
    public float calculateTotal () {
        float total = 0;
        Enumeration<String> keys = items.keys();

        while(keys.hasMoreElements()) {
            total  = total + items.get(keys.nextElement());
        }

        return total;
```

# SPLITTEREMPORARYVARIABLE




CÓDIGO SIN REFACTORIZAR

En el método totalPrice() se utiliza temp para almacenar el valor del impuesto al valor agregado vat calculado como (vat * price) / 100. Luego, se actualiza temp sumándole el precio original para obtener el precio total con el impuesto. Finalmente, se resta el descuento al valor de temp y se devuelve el resultado.

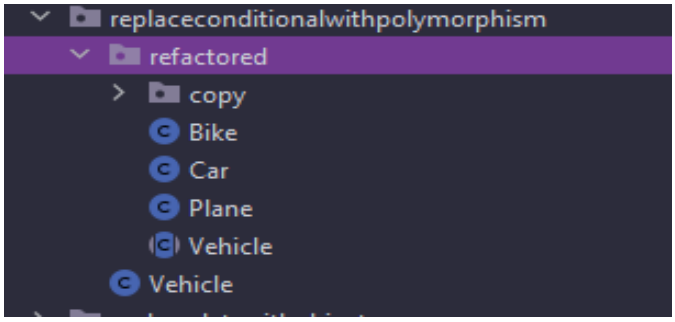
```
public class Invoice {
    1 usage
    public float totalPrice (float price, float vat, float discount) {
        float temp = 0;
        temp = (vat * price) / 100;
        System.out.println("Applied vat: " + temp);
        temp = price + temp;
        System.out.println("Total with vat: " + temp);
        return temp - discount;
    }
```

CÓDIGO REFACTORIZADO

Se ha creado una variable appliedVat que representa el impuesto al valor agregado aplicado y calcula (vat * 100) / price. Se utiliza appliedVat directamente para imprimir el valor del impuesto aplicado y para calcular el precio total con el impuesto.

```
public class Invoice {

    no usages
    public float totalPrice (float price, float vat, float discount) {
        float appliedVat = (vat * 100) / price;
        System.out.println("Applied vat: " + appliedVat);

        float priceWithVat = price + appliedVat;
        System.out.println("Total: " + priceWithVat);

        return priceWithVat - discount;
    }

    /*
     * Another Step
    public float totalPrice (float price, float vat, float discount) {
        return price + appliedVat(price, vat) - discount;
    }

    private float appliedVat (float price, float vat) {
        return (vat * price) / 100;
    }*/
}
```

# REPLACECONDITIONALWITHPOLYMORPHISM



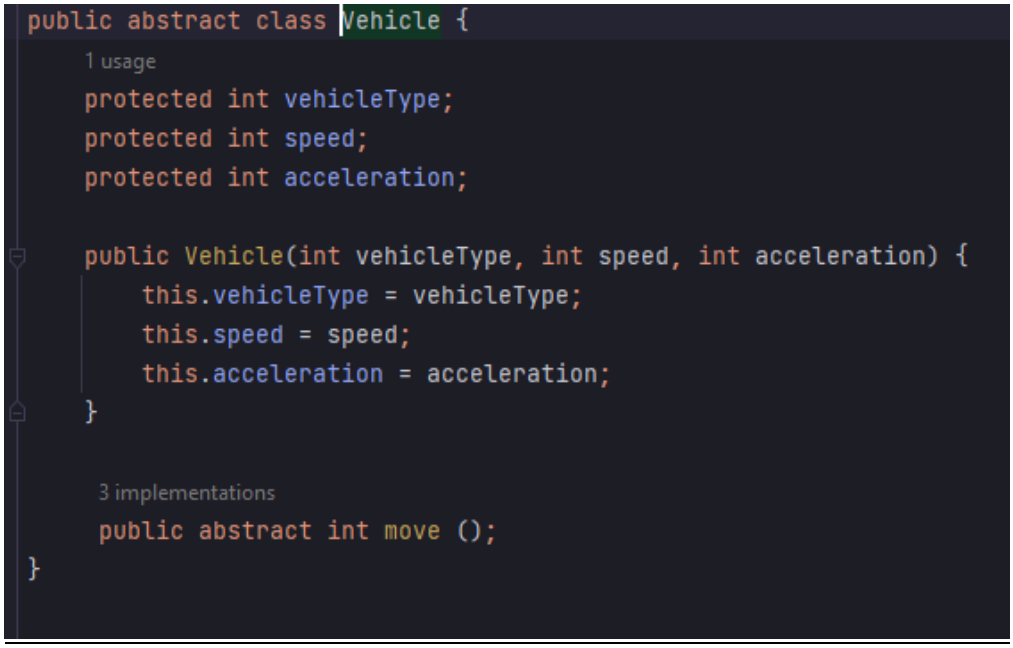

CÓDIGO SIN REFACTORIZAR

```
public class Vehicle {

    1 usage
    private static final int CAR = 0;
    1 usage
    private static final int BIKE = 1;
    1 usage
    private static final int PLANE = 2;
    2 usages
    private int vehicleType;
    3 usages
    private int speed;
    3 usages
    private int acceleration;

    public Vehicle(int vehicleType, int speed, int acceleration) {
        this.vehicleType = vehicleType;
        this.speed = speed;
        this.acceleration = acceleration;
    }

    public int move () {
          int result = 0;
          switch (vehicleType) {
            case CAR:
                      result = speed * acceleration * 5;
                      break;
            case BIKE:
                      result = speed * 10;
                      break;
            case PLANE:
                      result = acceleration * 2;
                      break;
          }
          return result;
        }
}
```

El método move() utiliza un switch para calcular la distancia recorrida según el tipo de vehículo.

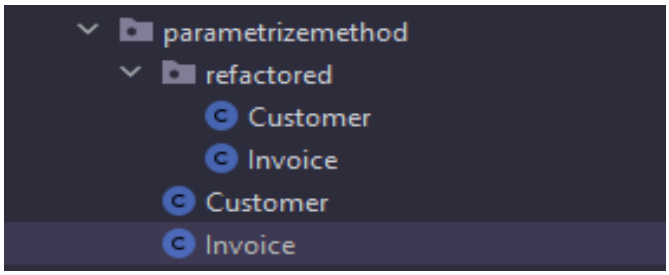
CÓDIGO REFACTORIZADO

El método move() es abstracto en la clase Vehicle, lo que obliga a las clases derivadas a proporcionar una implementación. Se ha creado la clase Bike que hereda de Vehicle y proporciona una implementación específica del método move() para el tipo de vehículo BIKE.



```
public abstract class Vehicle {
    1 usage
    protected int vehicleType;
    protected int speed;
    protected int acceleration;

    public Vehicle(int vehicleType, int speed, int acceleration) {
        this.vehicleType = vehicleType;
        this.speed = speed;
        this.acceleration = acceleration;
    }

    3 implementations
    public abstract int move ();
}
```

# PARAMETRIZEMETHOD



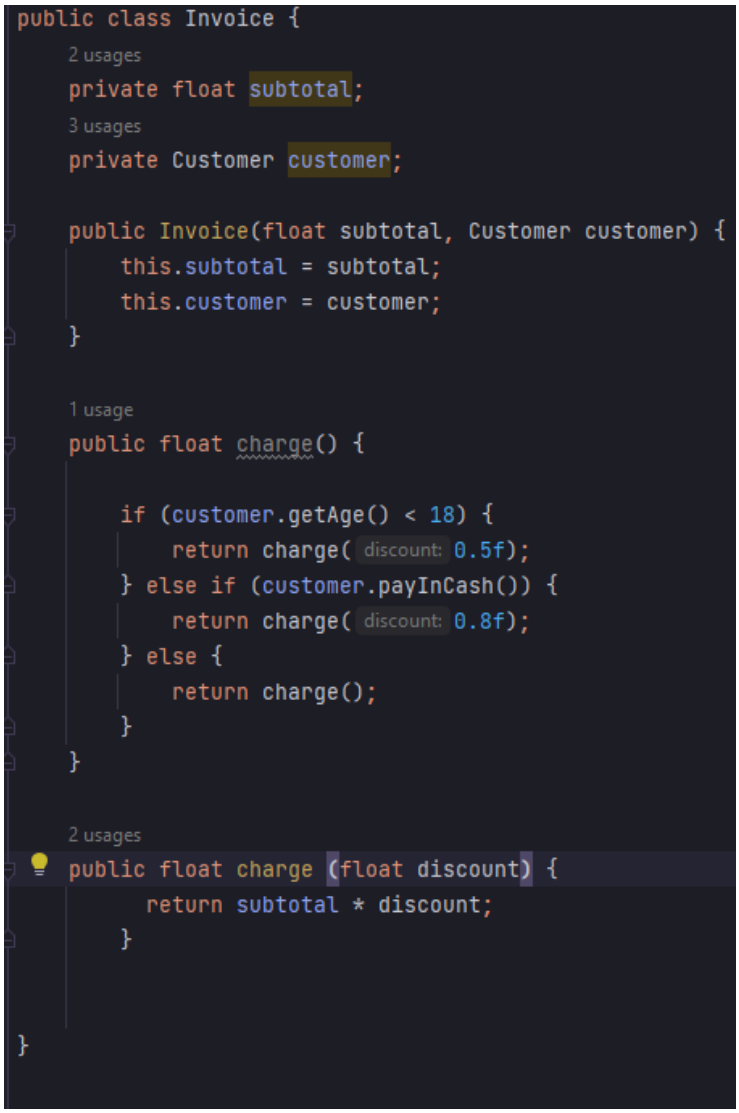

CÓDIGO SIN REFACTORIZAR

```java
public class Invoice {
    4 usages
    private float subtotal;
    3 usages
    private Customer customer;

    public Invoice(float subtotal, Customer customer) {
        this.subtotal = subtotal;
        this.customer = customer;
    }

    3 usages
    public float charge() {
        if (customer.getAge() < 18) {
            return chargeWithUnderageDiscount();
        } else if (customer.payInCash()) {
            return chargeWithCashDiscount();
        } else {
            return chargeNormal();
        }
    }

    1 usage
    private float chargeWithUnderageDiscount() {
        float total = subtotal * 0.5f;
        return total;
    }

    1 usage
    private float chargeWithCashDiscount() {
        float total = subtotal * 0.8f;
        return total;
    }

    1 usage
    private float chargeNormal() { return subtotal; }
}
```

El método charge() calcula el total de la factura según la edad del cliente y si paga en efectivo. Dentro de charge() se ejecuta chargeWithCashDiscount() y chargeNormal() para calcular los totales con descuento.

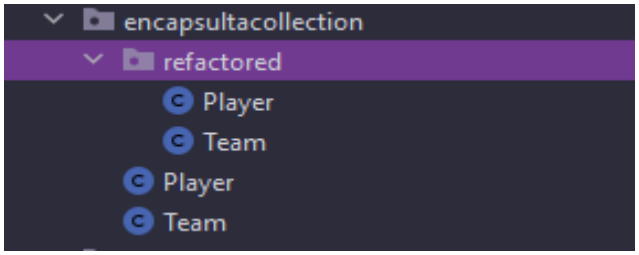
CÓDIGO REFACTORIZADO

El método charge(float discount) toma un parámetro discount que indica el descuento a aplicar al subtotal. charge(), determina el descuento según la edad del cliente y si paga en efectivo. charge(float discount) calcula total de la factura multiplicando el subtotal por el descuento y lo retorna.



```
public class Invoice {
    2 usages
    private float subtotal;
    3 usages
    private Customer customer;

    public Invoice(float subtotal, Customer customer) {
        this.subtotal = subtotal;
        this.customer = customer;
    }

    1 usage
    public float charge() {

        if (customer.getAge() < 18) {
            return charge( discount: 0.5f);
        } else if (customer.payInCash()) {
            return charge( discount: 0.8f);
        } else {
            return charge();
        }
    }

    2 usages
    public float charge (float discount) {
          return subtotal * discount;
        }


}
```

# ENCAPSULATECOLLECTION




CÓDIGO SIN REFACTORIZAR

```
public class Team {
    2 usages
    private String name;
    2 usages
    private Date creation;
    2 usages
    private ArrayList<Player> players = new ArrayList<~>();

    1 usage
    public Team(String name, Date creation) {
        this.name = name;
        this.creation = creation;
    }

    public String getName() {
        return name;
    }

    no usages
    public Date getCreation() {
        return creation;
    }

    no usages
    public ArrayList<Player> getPlayers() {
        return players;
    }

    1 usage
    public int totalPlayers() {
        return players.size();
    }
}
```

CÓDIGO REFACTORIZADO

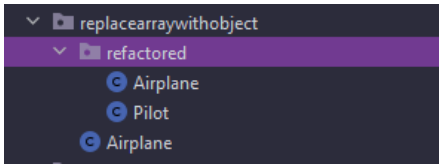
El método getPlayer(int index) para obtener un jugador específico de la lista, según su índice.

El método addPlayer(Player player) para agregar un jugador a la lista.

El método removePlayer(int index) para eliminar un jugador de la lista, según su índice..

```
public Team(String name, Date creation) {
    this.name = name;
    this.creation = creation;
}

public String getName() {
    return name;
}

no usages
public Date getCreation() {
    return creation;
}

no usages
public Player getPlayer (int index) {
    return players.get(index);
}

no usages
public void addPlayer (Player player) {
    players.add(player);
}

no usages
public void  removePlayer (int index) {
    players.remove(index);
}

no usages
public int totalPlayers() {
    return players.size();
}
```

# REPLACEARRAYWITHOBJECT




CÓDIGO SIN REFACTORIZAR

pilotData es un array de String con un tamaño de 3 que almacena los datos del piloto.

El método initPilot() se utiliza para inicializar los datos del piloto en el array pilotData.

El método toString() muestra el modelo del avión y el nombre del piloto almacenado en pilotData[0].

```
public class Airplane {

    2 usages
    private String model;
    4 usages
    private String pilotData[] = new String[3];

    1 usage
    public Airplane(String model) { this.model = model; }

    1 usage
    public void initPilot(String name, String license, int flightHours) {
        pilotData[0] = name;
        pilotData[1] = license;
        pilotData[2] = Integer.toString(flightHours);
    }

    @Override
    public String toString() { return "Airplane [model=" + model + ", pilot=" + pilotData[0] + "]"; }
}
```

CÓDIGO REFACTORIZADO

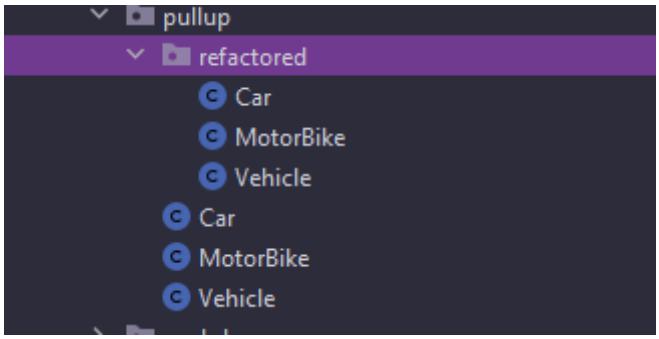
Se ha reemplazado el array pilotData con una instancia de la clase Pilot.

El método initPilot() instancia de Pilot y tiene de parámetros las variables del la clase Pilot

El método toString() muestra el modelo del avión y al piloto

```
public class Airplane {

    2 usages
    private String model;
    2 usages
    private Pilot pilot;

    no usages
    public Airplane(String model) {
        this.model = model;
    }

    no usages
    public void initPilot(String name, String license, int flightHours) {
        pilot = new Pilot(name, license, flightHours);
    }

    @Override
    public String toString() {
        return "Airplane [model=" + model + ", pilot=" + pilot + "]";
    }
```

# PULLUP




CÓDIGO SIN REFACTORIZAR

```
2 inheritors
public class Vehicle {
    protected String name;
}
```

```
public class Car extends Vehicle {
    no usages
    private String plate;
    no usages
    private String trunk;
    1 usage
    private boolean isTrunkOpened;

    no usages
    public void start() {
    }

    no usages
    public boolean isTrunkOpen() {
        return isTrunkOpened;
    }
```

```
no usages
public class MotorBike extends Vehicle {
    no usages
    private String plate;
    no usages
    private String helmet;

    no usages
    public void start() {
    }
}
```

CÓDIGO REFACTORIZADO

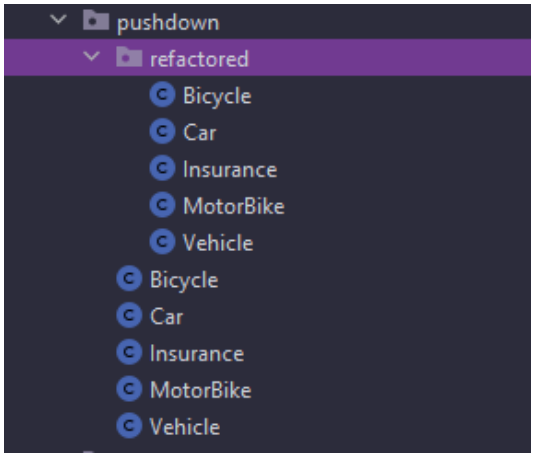
Se crea la variable método start() en Vehicle para luego usarlas en las demás clases que son heredaras de Vehicle (MotorBike y Car). Esto se hace para eliminar la duplicación de código y usar la herencia de las clases.

```
2 inheritors
public class Vehicle {
    protected String name;
    no usages
    protected String plate;

    no usages
    public void start() {
    }
}
```

```
n usages
public class MotorBike extends Vehicle {
    no usages
    private String helmet;
}
```

```
public class Car extends Vehicle {
    no usages
    private String trunk;
    1 usage
    private boolean isTrunkOpened;

    no usages
    public boolean isTrunkOpen() {
        return isTrunkOpened;
    }
}
```

# PUSHDOWN




CÓDIGO SIN REFACTORIZAR

```
public class Vehicle {
    protected String name;
    no usages
    protected String plate;
    no usages
    protected Insurance insurance;

    no usages  1 override
    public void start() {
    }
}
```

```
public class MotorBike extends Vehicle {
    no usages
    private String type;

}
```

```
public class Car extends Vehicle {
    no usages
    private String trunk;
    1 usage
    private boolean isTrunkOpened;

    no usages
    public void start() {
    }

    no usages
    public boolean isTrunkOpen() {
        return isTrunkOpened;
    }

}
```

```
6 usages
public class Insurance {

}
```

```
no usages
public class Bicycle extends Vehicle {
    no usages
    private String helmet;
}
```

CÓDIGO REFACTORIZADO

Vehicle tiene una variable name. La clase MotorBike hereda de Vehicle y agrega las variables de instancia type, plate y insurance, así como el método start(). La clase MotorBike ahora tiene acceso directo a la variable plate y la clase Insurance para evitar usar dos veces el código en la clase base. Además, la clase Car y la clase Bicycle no se modifican en este caso, ya que no hay elementos que se deban descender en ellas.

```
3 heritors
public class Vehicle {
    protected String name;
}
```

```
no usages
public class Bicycle extends Vehicle {
    no usages
    private String helmet;
}
```

```
public class MotorBike extends Vehicle {
    no usages
    private String type;
    no usages
    protected String plate;
    no usages
    protected Insurance insurance;

    no usages
    public void start() {
    }
}
```

```
6 usages
public class Insurance {

}
```

```
public class Car extends Vehicle {
    no usages
    private String trunk;
    1 usage
    private boolean isTrunkOpened;
    no usages
    protected String plate;
    no usages
    protected Insurance insurance;


    no usages
    public boolean isTrunkOpen() { return isTrunkOpened; }

    no usages
    public void start() {
    }
```