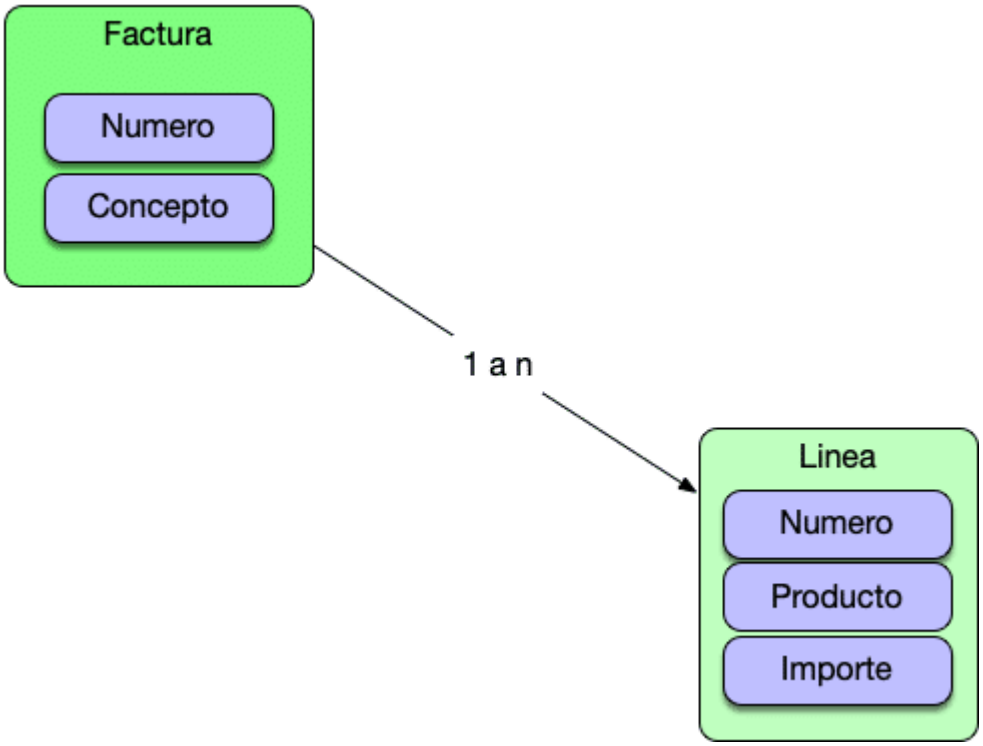
El principio KISS o (Keep It Simple Stupid) es uno de los principios de ingeniería de Software que hablan sobre cómo mantener la simplicidad dentro de nuestro código. Este concepto en muchas ocasiones cuesta entenderlo ya que mantener la simplicidad es un tema muy abierto . Vamos a ver cómo podemos construir un ejemplo que ayude a clarificarlo.

continua artículo premium ..

[ihc-hide-content ihc_mb_type=”show” ihc_mb_who=”4″ ihc_mb_template=”1″]

El principio KISS

Para ello vamos a crearnos dos clases Factura y Linea de Factura .



Ambas clases se encuentran relacionadas con una relación de 1 a N una (Factura contiene varias lineas).

```
package com.arquitecturajava;

import java.util.ArrayList;
import java.util.List;

public class Factura {

  private int numero;
  private String concepto;

  private List<Linea> lineas = new ArrayList<Linea>();

  public List<Linea> getLineas() {
    return lineas;
  }

  public void setLineas(List<Linea> lineas) {
    this.lineas = lineas;
  }

  public int getNumero() {
    return numero;
  }

  public void setNumero(int numero) {
    this.numero = numero;
```

```
}

public String getConcepto() {
  return concepto;
}

public void setConcepto(String concepto) {
  this.concepto = concepto;
}



public Factura(int numero, String concepto) {
  super();
  this.numero = numero;
  this.concepto = concepto;

}

public void addLinea(Linea linea) {

  lineas.remove(linea);
}

public void removeLinea(Linea linea) {

  lineas.add(linea);
}

@Override
public int hashCode() {
```

```
    final int prime = 31;
    int result = 1;
    result = prime * result + numero;
    return result;
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    Factura other = (Factura) obj;
    if (numero != other.numero)
      return false;
    return true;
  }

}



package com.arquitecturajava;

public class Linea {

  private int numero;
  private String producto;
  private double importe;
```

```
public int getNumero() {
  return numero;
}
public void setNumero(int numero) {
  this.numero = numero;
}
public String getProducto() {
  return producto;
}
public void setProducto(String producto) {
  this.producto = producto;
}
public double getImporte() {
  return importe;
}
public void setImporte(double importe) {
  this.importe = importe;
}
public Linea(int numero, String producto, double importe) {
  super();
  this.numero = numero;
  this.producto = producto;
  this.importe = importe;
}
@Override
public int hashCode() {
  final int prime = 31;
  int result = 1;
  result = prime * result + numero;
  return result;
}
```

```
  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    Linea other = (Linea) obj;
    if (numero != other.numero)
      return false;
    return true;
  }
}
```

Facturas y Totales

Disponemos de ambas clases y queremos calcular cuál es el importe total de la Factura y el importe total con IVA. Se trata de cambios sencillos en las clases ya que nos valdrá con añadir el método getImporteConIVA() en la Linea :

```
public double getImporteConIVA() {
    return importe * 1.21;
  }
```

Y añadir los de calculo del total en las Facturas

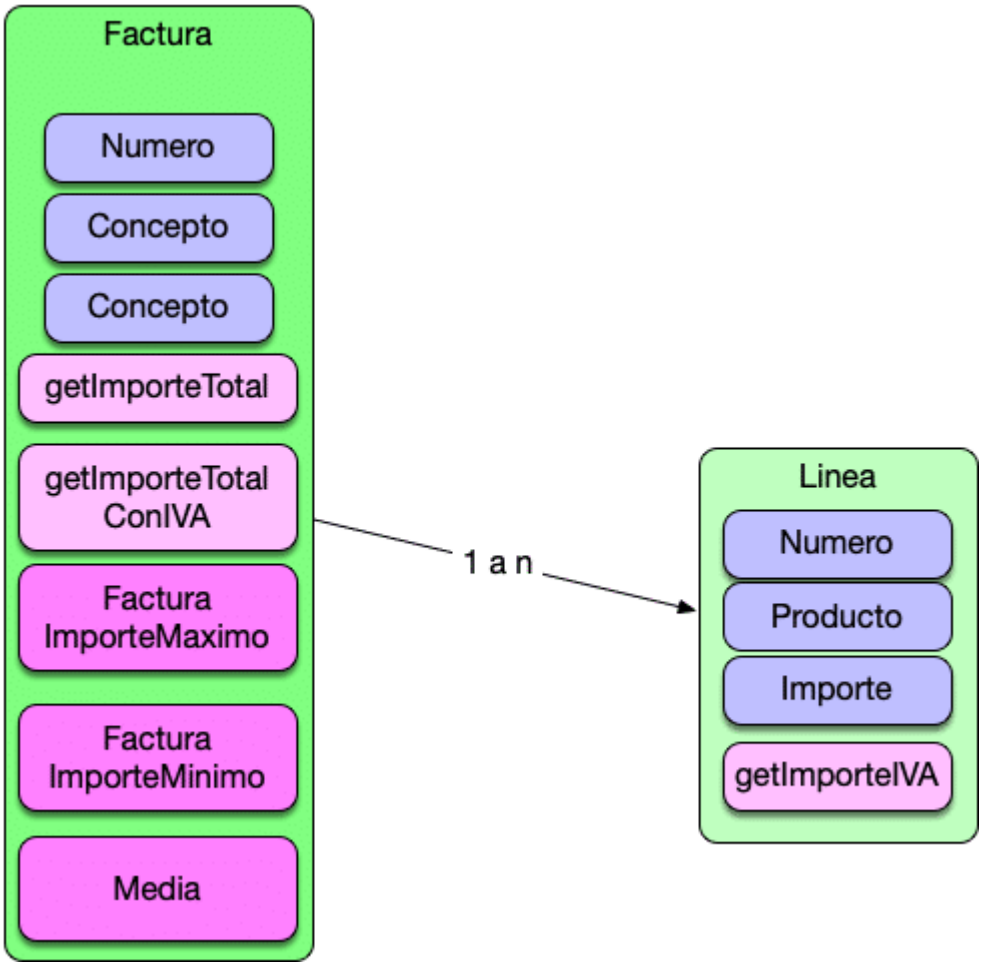
```
public double getImporte() {

    double total = 0;
    for (Linea l : lineas) {
```

```
    total += l.getImporte();
  }
  return total;
}

public double getImporteConIVA() {

  double total = 0;
  for (Linea l : lineas) {

    total += l.getImporteConIVA();
  }
  return total;
}
```

Todo es correcto y la implementación yo diría que es bastante trivial acabamos de dividir correctamente las responsabilidades entre clases.

Facturas y Cálculos estadísticos

En muchas ocasiones al tratarse de una clase tan importante como la clase Factura , nos podemos encontrar con que necesitamos realizar unos cálculos estadísticos sobre un grupo de Facturas .Por ejemplo calcular la Media , la Factura con importe máximo, la Factura con importe mínimo o cosas similares. ¿Cómo podemos abordar esto? . Bueno en principio es relativamente sencillo ya que se trata de cálculos que afectan a un conjunto de Facturas . Podemos construir métodos estáticos en la clase y que reciban una lista de Facturas.

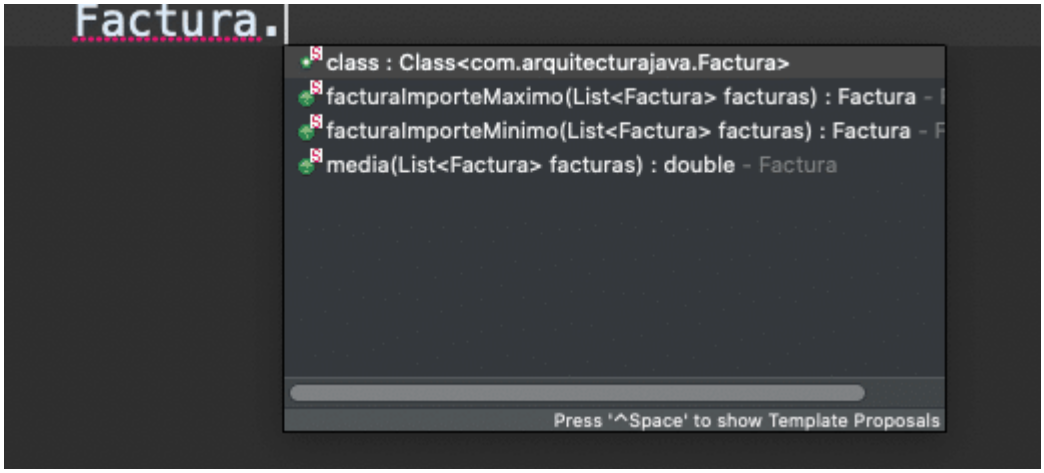


Veamos el código:

```
public static Factura facturaImporteMaximo(List<Factura> facturas) {

    Factura maxima=facturas.get(0);
    for (Factura f : facturas) {

      if(f.getImporte()>maxima.getImporte()) {
        maxima=f;
```

```
    }
  }
  return maxima;
}
public static Factura facturaImporteMinimo(List<Factura> facturas) {

  Factura minimo=facturas.get(0);
  for (Factura f : facturas) {

    if(f.getImporte()<minimo.getImporte()) {
      minimo=f;
    }
  }
  return minimo;
}
public static double media(List<Factura> facturas) {

  double total = 0;
  for (Factura f : facturas) {

    total+=f.getImporte();
  }
  return total/facturas.size();
}
```

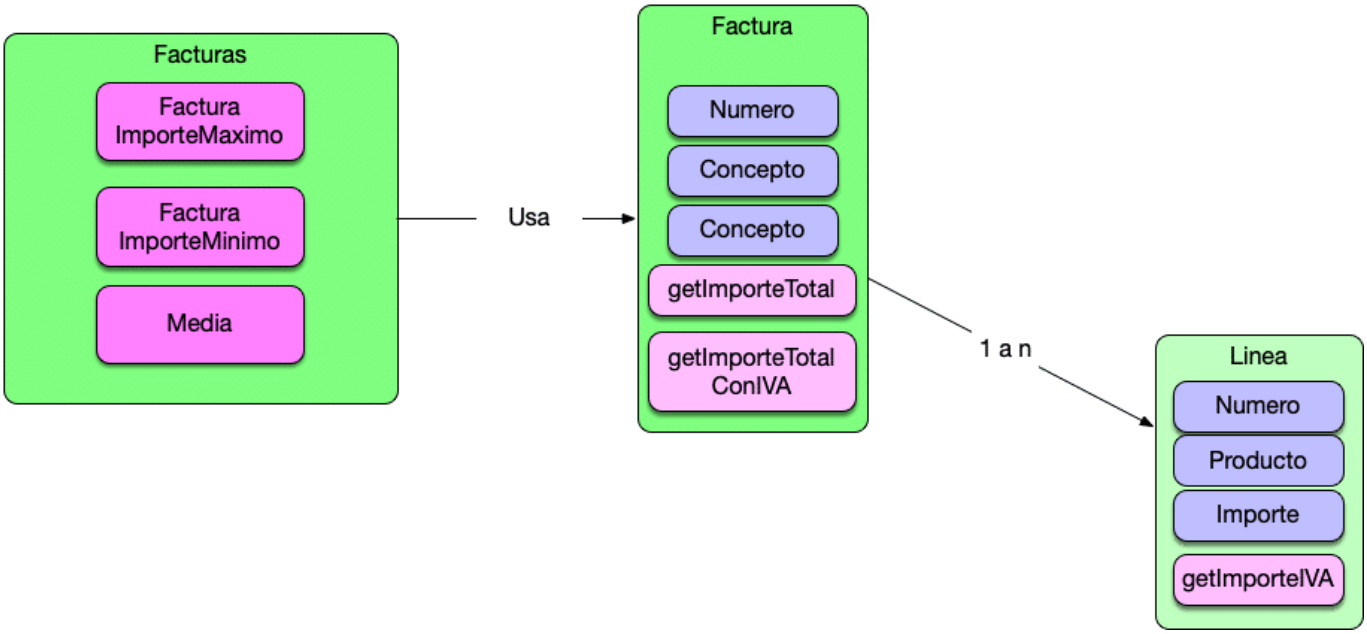
Así pues cuando nosotros usemos la clase Factura podremos acceder a los métodos estáticos y realizar los cálculos estadísticos que correspondan.



La pregunta es si es esto lo correcto. La realidad es que todo depende mucho, ya que si nos encontramos ante un ejemplo relativamente pequeño podría ser una opción válida . Sin embargo es fácil preguntarse qué pasará cuando la clase tenga que ubicar un mayor conjunto de métodos estadísticos , estos pueden ser fácilmente decenas. En este caso estamos complicando el manejo de clases muy habituales como Factura y Linea a cualquiera de los desarrolladores . No estamos utilizando el principio KISS que nos dice que mantengamos la estructura de nuestro código sencilla.

El principio KISS al rescate

Podemos reflexionar sobre cómo tenemos construido este código y valorar si quizás las responsabilidades que estamos asignando a la clase Factura son demasiadas. Desde mi punto de vista es así y deberíamos separarlas y aislarlas en una clase diferente que se encargue de esa gestión de métodos estadísticos. Por lo tanto una implementación más correcta y más cercana a el uso del principio KISS sería crear una clase “Facturas” en plural y allí almacenar dichos métodos.



Veamoslo:

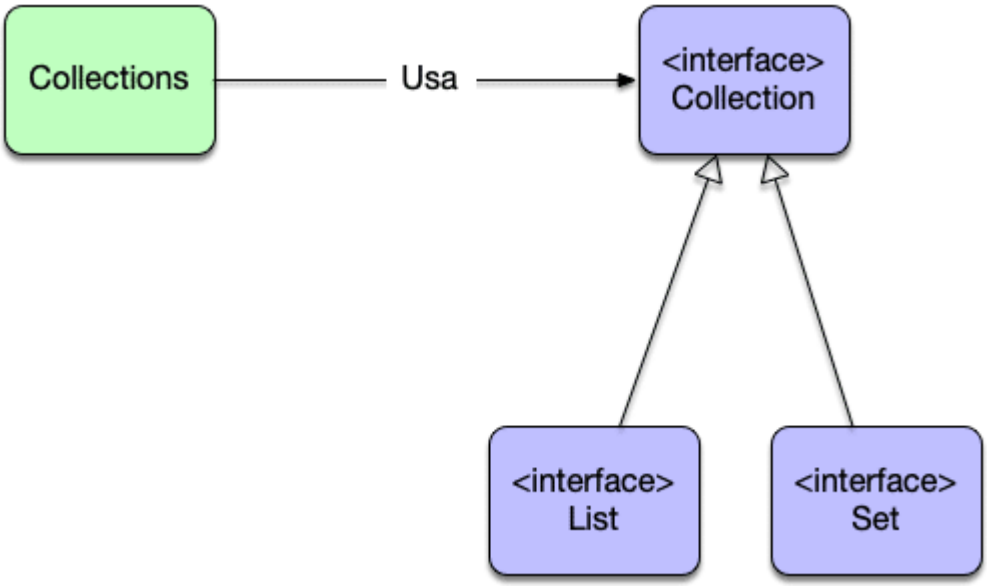
```
package com.arquitecturajava;

import java.util.List;

public class Facturas {

  public static Factura facturaImporteMaximo(List<Factura> facturas) {

    Factura maxima=facturas.get(0);
    for (Factura f : facturas) {

      if(f.getImporte()>maxima.getImporte()) {
        maxima=f;
      }
    }
    return maxima;
```

```
  }
  public static Factura facturaImporteMinimo(List<Factura> facturas) {

    Factura minimo=facturas.get(0);
    for (Factura f : facturas) {

      if(f.getImporte()<minimo.getImporte()) {
        minimo=f;
      }
    }
    return minimo;
  }
  public static double media(List<Factura> facturas) {

    double total = 0;
    for (Factura f : facturas) {

      total+=f.getImporte();
    }
    return total/facturas.size();
  }
}
```

De esta manera dividimos las responsabilidades entre las diferentes clases y mantenemos el código de las clases Factura y Linea sencillo y manejable.

Java APIS y el principio KISS

Un ejemplo sencillo de esta forma de separar responsabilidades nos lo podemos encontrar en el API de Java en donde el framework de colecciones mantiene un conjunto de métodos bastante compacto a nivel de interfaces List , Map y Set delegando en la clase Collections para realizar todo tipo de operaciones adicionales.



Otros artículos relacionados

1. Java Collections Remove con Java 8
2. Java List to Map y el uso de Collectors
3. Java Collections List vs Set (I)

[/ihc-hide-content]