Programación Modular y Orientación a Objetos - Laboratorio 3

Requisitos previos

El alumnado se desenvuelve en el entorno Eclipse y maneja adecuadamente la perspectiva Java; es capaz de crear y representar (mediante diagramas UML) clases sencillas y relaciones simples entre clases; y sabe realizar casos y suites de prueba con el framework JUnit.

Objetivos

Este laboratorio se centrará en afianzar los conocimientos adquiridos en los laboratorios anteriores, así como en profundizar en la visión de la POO frente a la perspectiva de la programación imperativa. Además, se verán los tipos enumerados y el concepto de interfaz en Java, y se usará por vez primera la plataforma Web-CAT.

Al finalizar este laboratorio, el alumnado deberá ser capaz de:

- Desenvolverse en el uso del entorno Eclipse para la realización de ejercicios sencillos, que se irán complicando en sucesivos laboratorios, desde una perspectiva orientada a objetos.
- Comprender el concepto de interfaz, y ser capaz de crear una clase que implemente una interfaz dada.
- Entender la perspectiva de la orientación a objetos, comprender los diagramas de clase UML, y utilizar JUnit para crear pruebas unitarias.
- Conectarse a su cuenta Web-CAT y subir sus clases y casos de prueba para que las implementaciones sean corregidas de modo instantáneo y automático.

Motivación

Antes de comenzar a trabajar con elementos más avanzados, como los TAD y las MAE, o los aspectos relativos a la herencia (que de momento solamente se han dejado entrever), es conveniente afianzar todo lo visto hasta ahora. Este laboratorio, que consiste en la realización de una batería de ejercicios sencillos que el alumnado debería poder completar sin mayores problemas, comienza con una distinción entre cómo se abordan los problemas desde la orientación a objetos en comparación con la programación imperativa tradicional.

Tarea 1: la clase Operacion (pensando en imperativo)

Una visión tradicional, esto es, en términos de la programación imperativa, está guiada por las operaciones que se realizan con los datos. Es por ello que, de cara a plantear operaciones sencillas entre enteros (suma, resta, producto, cociente o resto) una primera aproximación llevaría a diseñar una clase Operación, similar a la que muestra el diagrama UML de la siguiente figura. Como puede verse, esta clase tiene tres atributos: *operando1* y *operando2*, de tipo entero, y *operador*, del tipo enumerado Operador, cuyos valores posibles son, en este ejemplo, SUMA, RESTA, PRODUCTO, COCIENTE, y RESTO.

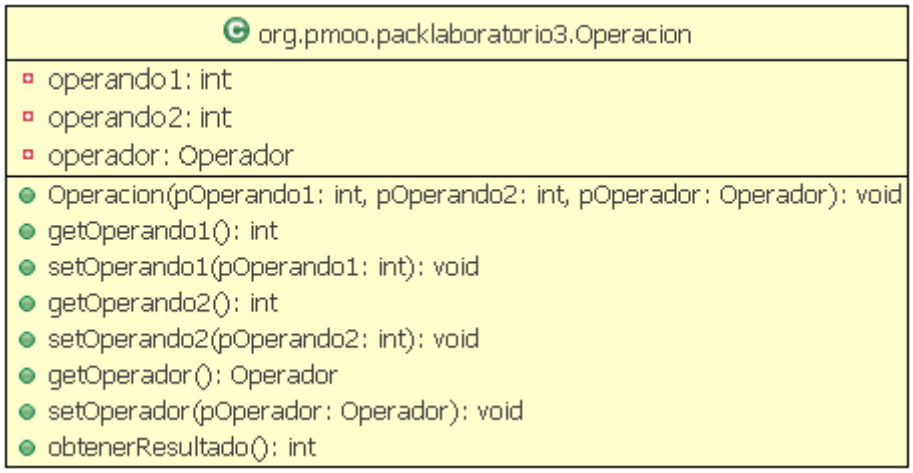

Figura 1 - Diagrama UML de la clase Operación.

Esta actividad servirá para plantear una primera solución, inspirada en la programación imperativa aunque utiliza objetos, para la implementación de operaciones entre números enteros. Sin embargo, en la siguiente tarea se podrá comprobar cómo se realiza esta labor más adecuadamente en el contexto de la orientación a objetos.

Se pide:

- Implementar la clase Operacion, incluyendo los tres citados atributos, la constructora (que en este caso recibe los valores de los atributos como parámetros), los *getters* y *setters*, y el método *obtenerResultado*(), que devolverá el resultado de realizar la operación (suma, resta, producto, cociente o resto de la división, en función del atributo *operador*) entre *operando1* y *operando2*.
- Implementar la clase OperacionTest con los casos de prueba, y ejecutarlos para verificar la corrección de los métodos de la clase Operacion.

Tarea 2: la clase Operando (pensando en OO)

La clave de la orientación a objetos es tener siempre presente que el diseño de las aplicaciones debe estar guiado por los objetos que participan, de manera que las operaciones que se pueden realizar con ellos quedan en un segundo plano, ya que se incluyen en los propios objetos. Dicho de otro modo, lo principal es identificar las clases, y si el diseño se ha realizado correctamente, los métodos (operaciones) estarán incluidos dentro de esas clases.

Para ilustrar esto, en esta actividad se pide realizar lo mismo que en la tarea anterior, solamente que con una perspectiva puramente orientada a objetos. Se espera que el alumnado comprenda y asimile la diferencia entre cómo se ha abordado la tarea anterior y lo que se solicita en ésta.

A la hora de plantear operaciones sencillas entre enteros (igual que antes, suma, resta, producto, cociente y resto de la división), la perspectiva orientada a objetos se fija en primer lugar en los elementos que participan: los operandos. Por este motivo, lo que habrá que definir será una clase Operando, similar a la que muestra la figura siguiente.

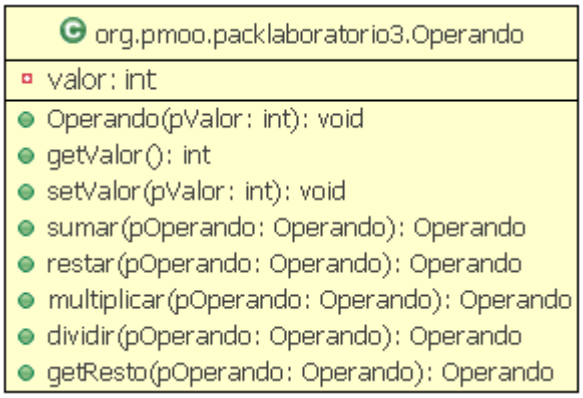

Figura 2 - Diagrama UML de la clase Operando.

Puede observarse que la clase Operando tiene un único atributo, que corresponde al valor (entero) del operando. Sin embargo, la mayor diferencia (y aquí radica la clave de la orientación a objetos) es que en lugar de un único método obtenerResultado() que realiza una operación u otra dependiendo del tipo de operación de que se trate, lo que tiene es un método por cada una de las operaciones que se pueda realizar con el operando. Nótese que el otro operando ya no es, como antes, un atributo de la clase, sino que se trata de otro objeto de tipo Operando que se pasa como parámetro al método que realiza la operación, que a su vez devuelve un objeto Operando cuyo valor es el resultado de la operación.

Se pide:

- Implementar la clase Operando, que solo tiene un atributo (valor), el *getter* y el *setter*, y los métodos *sumar*(), *restar*(), *multiplicar*(), *dividir*() y *getResto*(), que devuelven el objeto resultado de realizar la operación correspondiente entre el objeto actual (*this*) y el objeto que reciben como parámetro.
- Implementar los casos de prueba en la clase OperandoTest, y ejecutarlos para verificar la corrección de los métodos de la clase Operando.

Tarea 3: la interfaz IFraccion

Siguiendo con la filosofía de orientación a objetos, y evolucionando el ejercicio desarrollado en la tarea anterior, en esta ocasión se trabajará con operandos de tipo Fracción.

Se pide implementar la clase Fraccion, que tendrá dos atributos (el numerador y el denominador), sus correspondientes *getters* y *setters*, y los métodos que permitan realizar operaciones aritméticas (suma, resta, producto, cociente y simplificación de una fracción) y de comparación entre fracciones (igualdad, mayor, menor, mayor o igual, y menor o igual).

También habrá que implementar la clase FraccionTest, en la que se desarrollarán los casos de prueba necesarios para verificar la corrección de los métodos de la clase Fraccion.

La clase Fraccion deberá implementar la siguiente interfaz, que está disponible en formato electrónico junto a este enunciado:

```
package org.pmoo.packlaboratorio3;


public interface IFraccion
{
  public abstract int getNumerador();
  public abstract void setNumerador(int pNumerador);
  public abstract int getDenominador();
  public abstract void setDenominador(int pDenominador);
  public abstract void simplificar();
  public abstract Fraccion sumar (Fraccion pFraccion);
  public abstract Fraccion restar (Fraccion pFraccion);
  public abstract Fraccion multiplicar (Fraccion pFraccion);
  public abstract Fraccion dividir (Fraccion pFraccion);
  public abstract boolean esIgualQue(Fraccion pFraccion);
  public abstract boolean esMayorQue(Fraccion pFraccion);
  public abstract boolean esMenorQue(Fraccion pFraccion);
  public abstract boolean esMayorOIgualQue(Fraccion pFraccion);
  public abstract boolean esMenorOIgualQue(Fraccion pFraccion);
}
```

Figura 3 - Interfaz IFraccion que se pide implementar.

La constructora deberá recibir como parámetro los valores de las partes real e imaginaria del número complejo, por lo que su cabecera deberá ser similar a la siguiente:

public Fraccion (int pNumerador, int pDenominador).

Además, tanto la constructora como el método *setDenominador()* deberán impedir que se le asigne al atributo denominador el valor cero, y mostrarán un aviso por la consola del sistema cuando esto ocurra.

Aunque podría hacerse de otra manera, en esta tarea se va a suponer que el método *simplificar()* siempre deja el signo de la fracción en el numerador, de manera que al simplificar, por ejemplo, la fracción 4/-6 el resultado será la fracción -2/3, y no 2/-3. Por su parte, los métodos *sumar()*, *restar()*, *multiplicar()* y *dividir()*, una vez realizada la operación correspondiente, devolverán la fracción resultado simplificada. Así, por ejemplo, el resultado de sumar las fracciones 2/-3 y 2/6 será -1/3 y no -2/6.

Tarea 4: la interfaz IComplejo

Un número complejo tiene dos componentes numéricas: *a* (la parte real), y *b* (la parte imaginaria). Estas componentes se utilizan para representar los números complejos mediante coordenadas ortogonales (z = *a* + *b*i), aunque en ocasiones resulta más sencillo recurrir a las coordenadas polares, definidas en términos del ángulo y módulo, que se calculan de la siguiente forma:

- Ángulo: arctan(*b*/*a*)
- Módulo: $\sqrt{a^2 + b^2}$

Las operaciones entre números complejos se definen del siguiente modo:

- Suma: (a + bi) + (c + di) = (a+c) + (b+d)i
- Producto: (a + bi) * (c + di) = (a*c - b*d) + (ad+cb)i
- Igualdad: (a +bi) = (c + di) ↔ a=c && b=d

Para finalizar el laboratorio, se pide programar una clase NumeroComplejo que implemente la siguiente interfaz IComplejo (disponible en formato electrónico junto a este enunciado), así como la clase NumeroComplejoTest con los casos de prueba que verifiquen la corrección de los métodos desarrollados.

```
package org.pmoo.packlaboratorio3;
public interface IComplejo
{
  public abstract double getParteImaginaria();
  public abstract double getParteReal();
  public abstract double getAngulo();
   /*(devuelve el ángulo en grados, NO en radianes)*/
  public abstract double getModulo();
  public abstract IComplejo sumar(IComplejo pNumComp);
  public abstract IComplejo multiplicar(IComplejo pNumComp);
  public abstract boolean esIgual(IComplejo pNumComp);
}
```

Figura 4 - Interfaz IComplejo que se pide implementar.

La constructora deberá recibir como parámetro los valores de las partes real e imaginaria del número complejo, por lo que su cabecera deberá ser similar a la siguiente:

public NumeroComplejo (double pReal, double pImaginaria).