Análisis estático de código

PrInSo 2

Felipe Antonio Cano Galera

Contenido

Proyecto individual numero 2

En una primera instancia se puede comprobar en la ilustración 1 que los comentarios, que suele ser el calificativo de un buen codigo, no alcanzan ni el 1% en dos de las tres clases. Directamente en el App.java, que contiene el main, no hay un solo comentario.

Files in Java Project 'polinomio', Checkpoint 'Baseline' [Base Directory: 'C:\Users\felip\Desktop\imaguilaSourceMonitor-origin\Proyectos individua

File Name	Lines	Statements	% Branches	Calls	% Comments	Classes
src\examen2\App.java	107	81	28,4	69	0,0	1
src\examen2\Interpolacion.java	121	66	16,7	30	8,3	1
src\examen2\polinomio.java	286	192	33,9	80	0,7	1
test\examen2\InterpolacionTest.java	81	42	9,5	20	7,4	1
test\examen2\polinomioTest.java	255	188	2,1	101	7,5	1

Ilustración 1. Baja cantidad de comentarios

En “polinomio.java” se puede comprobar tambien que hay 0.7% de comentarios, lo cual es un problema, sobre todo cuando al comprobar el codigo se puede ver que los comentarios son 2 líneas muy simples (Ilustración 2)

```
public TreeMap<Integer, Double> ruffini(int poli1, int poli2) {
        TreeMap<Integer, Double> solucion = new TreeMap<>();
        TreeMap<Integer, Double> op1 = polinomios.get(poli1);
        double divisor = -1 * polinomios.get(poli2).get(0);
        // tiene que existir termino independiente ejemplo 3x2+2x -> 3x +2
        if (op1.get(0) == null || op1.get(0) == 0) {
                eliminarX(op1);
        }
        // tiene que tener 0 en los exponentes que no esten.
        for (int i = 0; i <= op1.lastKey(); i++) {
                if (op1.get(i) == null) {
                        op1.put(i, (double) 0);
                }
        }
```

Ilustración 2. Los dos únicos comentarios en polinomio.java

La ausencia de comentarios lleva a que métodos como “calculoRouth” que trabajan con matrices, algo que no es fácil de ver a simple vista al tener que imaginarse la estructura, no tengan ningún tipo de informacion previa. Además, considero el nombre un poco obtuso.

```
private double calculoRouth(int y, int x, Double[][] matriz) {
        double solucion = Double.NaN;
        if (matriz[y - 1][0] != 0) {
                solucion = ((matriz[y - 1][0] * matriz[y - 2][x + 1]) - (matriz[y - 2][0] * matriz[y - 1][x + 1]))
                                / matriz[y - 1][0];
        }
        return solucion;

}
```

Ilustración 3. Metodo "CalculoRouth"

Por último, interpolacion.java tiene un porcentaje un poco mas prometedor de comentarios, pero viendo el codigo se puede comprobar que la mitad son autogenerados, como se puede comprobar en la ilustración 4.

```
private static double calculoXr(double xizq2, double xder2) {
	// TODO Auto-generated method stub
	double resultado;
	double fdeIzq = sustitucion(xizq2);
	double fdeDer = sustitucion(xder2);
	double c = xder2 - (fdeDer / (fdeDer - fdeIzq)) * (xder2 - xizq2);

	return c;

}
```

Ilustración 4. Comentarios autogenerados que no aportan nada

```
public static double sustitucion(double punto) {
	double resultado = 0.0;

	TreeMap<Integer, Double> pol1 = new TreeMap<>();
	// f(x)=x 5 -x 4 +x 3 -3
	pol1.put(5, 1.0);
	pol1.put(4, -1.0);
	pol1.put(3, 1.0);
	pol1.put(2, 0.0);
	pol1.put(1, 0.0);
	pol1.put(0, -3.0);

	// polinomio = new polinomio(pol1);
	// this.polinomioActual = pol1;
	// this.polinomios = new ArrayList<>();
	// this.polinomios.add(polinomioActual);

	for (Iterator<Integer> it = pol1.keySet().iterator(); it.hasNext();) {
		int key = it.next();
		double coeficiente = pol1.get(key);
		int grado = key;
		if (coeficiente == 0) {
			resultado += 0;

		} else {
			resultado += coeficiente * (Math.pow(punto, grado));

		}
```

Ilustración 5. Codigo comentado

Y los otros comentarios no son más que código comentado.
Los test siguen la misma tónica, asi que se puede afirmar que la documentación de este codigo es inexistente.

Pasando a un aspecto mas técnico, en lo referente al codigo en si, en la clase App.java se tiene un altísimo porcentaje branches y es que cuando se comprueba en profundidad se pueden encontrar una gran cantidad de “if else”, como en la ilustración 6 y 7.

```
private static void operaciones(polinomio clase,int i) {
        Scanner leer=new Scanner(System.in);
        System.out.println("Lista de polinomios:");
        System.out.println(clase.listaPolinomios());
        System.out.print("Seleccione el 1º polinomio: ");
        int p1=Integer.parseInt(leer.nextLine());
        if(i==3){
                System.out.print("Seleccione un monomio: ");
        }else if(i==4){

        }else{
                System.out.print("Seleccione el 2º polinomio: ");
        }
        int p2=0;
        if(i!=4){
                p2=Integer.parseInt(leer.nextLine());
        }

        if(i==1){
        System.out.println("\n"+clase.poliToString(clase.getPolinomio(p1), "x")+"\n+\n"+clase.poliToString(clase.getPolinomio(p2), "x")+"\n----------------
        }else if(i==2){
                System.out.println("\n"+clase.poliToString(clase.getPolinomio(p1), "x")+"\n*\n"+clase.poliToString(clase.getPolinomio(p2), "x")+"\n---------
        }else if(i==3){
                System.out.println("\n"+clase.poliToString(clase.getPolinomio(p1), "x")+"\n/\n"+clase.poliToString(clase.getPolinomio(p2), "x")+"\n---------
        }else if(i==4){
                System.out.println("\n"+clase.routh(p1));
        }
        leer.nextLine();

}
```

Ilustración 6. Comprobación de % de branches en App.java

```
if("1".equals(opcion)){
        poli=new TreeMap<>();
        System.out.println("Se irán insertando bloques aXb.\na--> valor base.\nb-->valor exponente(no se pueden exponentes negativos).\nSi añade una base con el mismo exponente se sumaran l
        while(true){
                System.out.print("Inserte la base: ");
                double base=Double.parseDouble(leer.nextLine());
                System.out.print("Inserte el exponente: ");
                int exponente=Math.abs(Integer.parseInt(leer.nextLine()));

                if (poli.containsKey(exponente)) {
                        poli.replace(exponente, poli.get(exponente)+base);
                } else {
                        poli.put(exponente, base);
                }

                System.out.println("Su polinomio es "+clase.poliToString(poli, "x"));
                System.out.println("Para volver pulse 1.\nPara seguir añadiendo valores pulse cualquier tecla.");
                System.out.print("Opcion: ");

                if(leer.nextLine().equals("1")){
                        break;
                }
        }
        clase.add(poli);
}else if("2".equals(opcion)){
        while(true){
        System.out.println("1 - Suma de expresiones.");
        System.out.println("2 - Producto de expresiones.");
        System.out.println("3 - División método de Ruffini.");
        System.out.println("4 - Teorema de Routh-Hurwitz.");
        System.out.println("5 - Método de interpolación Lineal.");
        System.out.println("Cualquier tecla para volver.");
        System.out.print("Operación: ");
        String operacion=leer.nextLine();
        if(operacion.equals("1")){
                operaciones(clase,1);
        }else if(operacion.equals("2")){
                operaciones(clase,2);
        }else if(operacion.equals("3")){
                operaciones(clase,3);
        }else if(operacion.equals("4")){
                operaciones(clase,4);
        }else if(operacion.equals("5")){
                Interpolacion.main(args);
                Scanner aux=new Scanner(System.in);
                aux.nextLine();
        }else{
                break;
        }
        }
```

Ilustración 7. Comprobación de branches 2 en App.java

Todos esos else if se podrían sustituir por un switch, que dejaría el codigo mucho más ordenado y claro.

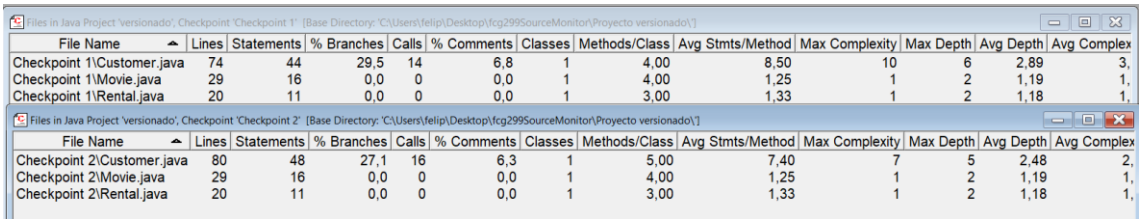
Además, concretando mas aún, se puede comprobar como en la ilustración 8 se está utilizando un numero para determinar a que else if se entra, para utilizar ese mismo numero para hacer una llamada, cuando podria hacerse directamente “operaciones(clase, operacion)” para los 4 primeros

```
if(operacion.equals("1")){
        operaciones(clase,1);
}else if(operacion.equals("2")){
        operaciones(clase,2);
}else if(operacion.equals("3")){
        operaciones(clase,3);
}else if(operacion.equals("4")){
        operaciones(clase,4);
}else if(operacion.equals("5")){
        Interpolacion.main(args);
        Scanner aux=new Scanner(System.in);
        aux.nextLine();
}else{
        break;
}
}
```

En las demas clases pasa lo mismo, mucho % de branches y todos con else if, cuando se debería hacer con switches.

Proyecto versionado

Del checkpoint 1 al 2



Files in Java Project 'versionado', Checkpoint 'Checkpoint 1' [Base Directory: 'C:\Users\felip\Desktop\fcg299SourceMonitor\Proyecto versionado\']

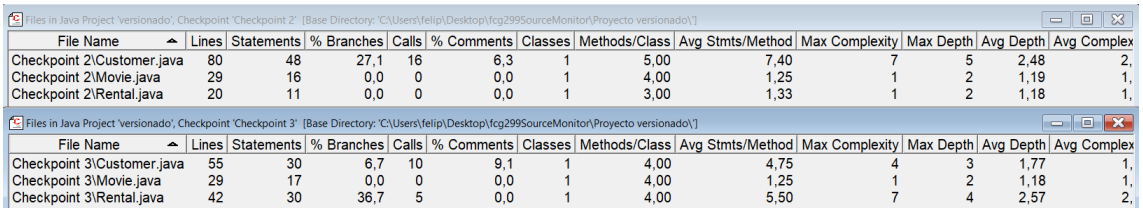
File Name	Lines	Statements	% Branches	Calls	% Comments	Classes	Methods/Class	Avg Stmts/Method	Max Complexity	Max Depth	Avg Depth	Avg Complex
Checkpoint 1\Customer.java	74	44	29,5	14	6,8	1	4,00	8,50	10	6	2,89	3,
Checkpoint 1\Movie.java	29	16	0,0	0	0,0	1	4,00	1,25	1	2	1,19	1,
Checkpoint 1\Rental.java	20	11	0,0	0	0,0	1	3,00	1,33	1	2	1,18	1,

Files in Java Project 'versionado', Checkpoint 'Checkpoint 2' [Base Directory: 'C:\Users\felip\Desktop\fcg299SourceMonitor\Proyecto versionado\']

File Name	Lines	Statements	% Branches	Calls	% Comments	Classes	Methods/Class	Avg Stmts/Method	Max Complexity	Max Depth	Avg Depth	Avg Complex
Checkpoint 2\Customer.java	80	48	27,1	16	6,3	1	5,00	7,40	7	5	2,48	2,
Checkpoint 2\Movie.java	29	16	0,0	0	0,0	1	4,00	1,25	1	2	1,19	1,
Checkpoint 2\Rental.java	20	11	0,0	0	0,0	1	3,00	1,33	1	2	1,18	1,

El cambio realizado del checkpoint 1 al 2 ha sido crear un metodo nuevo. Al ver de qué metodo se trata, se puede comprobar que se ha creado el metodo “amountOf” en “Customer.java”. Esta clase tenía un montón de codigo suelto, por lo que abstraer un poco de ese codigo en un metodo hace que sea reutilizable (que viene bien viendo que el codigo podria usarse varias veces) y aumenta su claridad

Del checkpoint 2 al 3



Files in Java Project 'versionado', Checkpoint 'Checkpoint 2' [Base Directory: 'C:\Users\felip\Desktop\fcg299SourceMonitor\Proyecto versionado\']

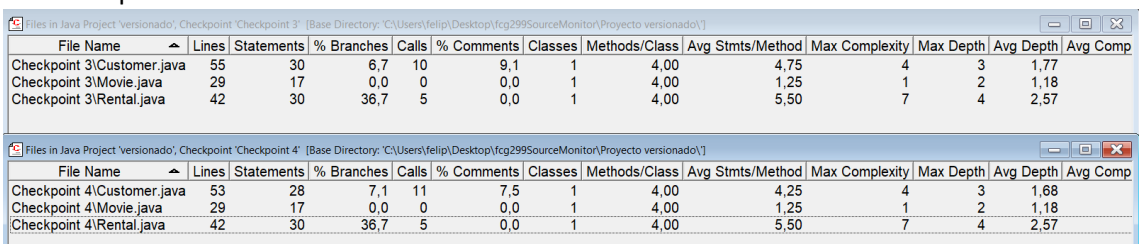
File Name	Lines	Statements	% Branches	Calls	% Comments	Classes	Methods/Class	Avg Stmts/Method	Max Complexity	Max Depth	Avg Depth	Avg Complex
Checkpoint 2\Customer.java	80	48	27,1	16	6,3	1	5,00	7,40	7	5	2,48	2,
Checkpoint 2\Movie.java	29	16	0,0	0	0,0	1	4,00	1,25	1	2	1,19	1,
Checkpoint 2\Rental.java	20	11	0,0	0	0,0	1	3,00	1,33	1	2	1,18	1,

Files in Java Project 'versionado', Checkpoint 'Checkpoint 3' [Base Directory: 'C:\Users\felip\Desktop\fcg299SourceMonitor\Proyecto versionado\']

File Name	Lines	Statements	% Branches	Calls	% Comments	Classes	Methods/Class	Avg Stmts/Method	Max Complexity	Max Depth	Avg Depth	Avg Complex
Checkpoint 3\Customer.java	55	30	6,7	10	9,1	1	4,00	4,75	4	3	1,77	1,
Checkpoint 3\Movie.java	29	17	0,0	0	0,0	1	4,00	1,25	1	2	1,18	1,
Checkpoint 3\Rental.java	42	30	36,7	5	0,0	1	4,00	5,50	7	4	2,57	2,

Del checkpoint 2 al 3 se ha eliminado el metodo “amount of” de customer.app y se ha creado “charge” en rental.app. Esto supone una mejora en el esquema del proyecto, dado que se tenia una clase muy grande y se ha separado uno de los métodos para llevarlo a una clase donde esta mejor ubicada. El metodo charge permite saber el coste de una factura, asi que tiene sentido que se ubique en la clase alquiler.

Del checkpoint 3 al 4



Files in Java Project 'versionado', Checkpoint 'Checkpoint 3' [Base Directory: 'C:\Users\felip\Desktop\fcg299SourceMonitor\Proyecto versionado\']

File Name	Lines	Statements	% Branches	Calls	% Comments	Classes	Methods/Class	Avg Stmts/Method	Max Complexity	Max Depth	Avg Depth	Avg Comp
Checkpoint 3\Customer.java	55	30	6,7	10	9,1	1	4,00	4,75	4	3	1,77	
Checkpoint 3\Movie.java	29	17	0,0	0	0,0	1	4,00	1,25	1	2	1,18	
Checkpoint 3\Rental.java	42	30	36,7	5	0,0	1	4,00	5,50	7	4	2,57	

Files in Java Project 'versionado', Checkpoint 'Checkpoint 4' [Base Directory: 'C:\Users\felip\Desktop\fcg299SourceMonitor\Proyecto versionado\']

File Name	Lines	Statements	% Branches	Calls	% Comments	Classes	Methods/Class	Avg Stmts/Method	Max Complexity	Max Depth	Avg Depth	Avg Comp
Checkpoint 4\Customer.java	53	28	7,1	11	7,5	1	4,00	4,25	4	3	1,68	
Checkpoint 4\Movie.java	29	17	0,0	0	0,0	1	4,00	1,25	1	2	1,18	
Checkpoint 4\Rental.java	42	30	36,7	5	0,0	1	4,00	5,50	7	4	2,57	

Los cambios en este commit han sido nimios, cambiando solo 2 statements

Del checkpoint 4 al 5

Files in Java Project 'versionado', Checkpoint 'Checkpoint 5' [Base Directory: 'C:\Users\felip\Desktop\fcg299SourceMonitor\Proyecto versionado\']

File Name	Lines	Statements	% Branches	Calls	% Comments	Classes	Methods/Class	Avg Stmts/Method	Max Complexity	Max Depth	Avg Depth	Avg Comp
Checkpoint 5\Customer.java	48	26	3,8	9	4,2	1	4,00	3,75	2	3	1,58	
Checkpoint 5\Movie.java	29	17	0,0	0	0,0	1	4,00	1,25	1	2	1,18	
Checkpoint 5\Rental.java	50	35	37,1	8	0,0	1	5,00	5,20	7	4	2,46	

Files in Java Project 'versionado', Checkpoint 'Checkpoint 4' [Base Directory: 'C:\Users\felip\Desktop\fcg299SourceMonitor\Proyecto versionado\']

File Name	Lines	Statements	% Branches	Calls	% Comments	Classes	Methods/Class	Avg Stmts/Method	Max Complexity	Max Depth	Avg Depth	Avg Comp
Checkpoint 4\Customer.java	53	28	7,1	11	7,5	1	4,00	4,25	4	3	1,68	
Checkpoint 4\Movie.java	29	17	0,0	0	0,0	1	4,00	1,25	1	2	1,18	
Checkpoint 4\Rental.java	42	30	36,7	5	0,0	1	4,00	5,50	7	4	2,57	

A primera vista, podemos comprobar que hay un cambio significativo en las lineas de codigo, sobre todo de rental. Y es que si vemos en detalle podemos comprobar que se ha añadido un metodo nuevo, frequentRenterPoints

Metrics Details For File 'Checkpoint 4\Rental.java'

Parameter	Value
Average Block Depth	2,57
Average Complexity	2,50
Most Complex Methods in 1 Class(es):	Complexity, Statements, Max Depth, Calls
Rental.getCharge()	7, 18, 4, 5
Rental.getDaysRented()	1, 1, 2, 0
Rental.getMovie()	1, 1, 2, 0
Rental.Rental()	1, 2, 2, 0

Block Depth	Statements
0	2
1	6

Kiviat Graph: % Comments, Methods/Class, Avg Stmts/Method, Max Complexity, Max Depth, Avg Depth, Avg Complexity

Block Histogram (statements vs. depth)

Help Copy Print Done

Metrics Details For File 'Checkpoint 5\Rental.java'

Parameter	Value
Average Block Depth	2,46
Average Complexity	2,80
Most Complex Methods in 1 Class(es):	Complexity, Statements, Max Depth, Calls
Rental.getCharge()	7, 18, 4, 5
Rental.getDaysRented()	1, 1, 2, 0
Rental.getFrequentRenterPoints()	4, 4, 2, 3
Rental.getMovie()	1, 1, 2, 0
Rental.Rental()	1, 2, 2, 0

Block Depth	Statements
0	2

Kiviat Graph: % Comments, Methods/Class, Avg Stmts/Method, Max Complexity, Max Depth, Avg Depth, Avg Complexity

Block Histogram (statements vs. depth)

Help Copy Print Done

En términos generales, el proyecto tiene unos nombres de métodos claros, aunque sigue teniendo líneas de codigo sueltas en Customer.app, que está metido en un while, que ni siquiera es un main.
A pesar de tener los métodos con nombres claros, hay una ausencia generalizada de documentación que debería abordarse, sobre todo si la aplicación va a crecer en complejidad.