LABORATORIO #2

JULIAN CAMILO LOPEZ BARRERO

JUAN SEBASTIAN PUENTES JULIO

ESCUELA COLOMBIANA DE INGENIERÍA JULIO GARAVITO

POOB

BOGOTÁ,COLOMBIA

14 DE FEBRERO DE 2025

Conociendo el proyecto [En lab02.doc]

1. El proyecto "graphCalculator" contiene una construcción parcial del sistema. Revisen el directorio donde se encuentra el proyecto. Describan el contenido en términos de directorios
y de las extensiones de los archivos.

Visualizando el directorio donde se encuentra el proyecto, se observa a simple vista la existencia de tres clases. Además, existen los diferentes archivos que componen la estructura del proyecto, como los .java donde se escribe el respectivo código de la clase y su respectiva documentación.

2. Exploren el proyecto en BlueJ

¿Cuántas clases tiene?
Solo tiene dos clases que son Graphic Calculator y Graph

¿Cuál es la relación entre ellas?
La relación entre ellas es una relación fuerte en la cual la calculadora contiene a los grafos , se podría decir que los almacena.

¿Cuál es la clase principal de la aplicación?
La clase principal de la aplicación es la calculadora debido a que allí es donde se va a realizar todo el proceso y las operaciones que esta tiene.

¿Cómo la reconocen?
La reconocemos al mirar la relación podemos observar que esta apunta al grafo lo cual quiere decir como lo decíamos en el punto anterior que esta va a contener a la otra clase grafos.

¿Cuáles son las clases "diferentes"?
La clase diferente es el GraphTest la podemos identificar debido a su color y a su nombre.

¿Cuál es su propósito?
El propósito es realizar distintas operaciones como la unión, intersección, junta y diferencia.
Para las siguientes dos preguntas sólo consideren las clases "normales":

3. Generen y revisen la documentación del proyecto: ¿está completa la documentación de cada clase? (Detallen el estado de documentación: encabezado y métodos)
Generando la documentación de las clases propuestas, no hay alguna que explique el objetivo o la finalidad del método y la clase. Falta documentar los métodos y así mismo la finalidad de la clase.

4. ***Revisen las fuentes del proyecto, ¿en qué estado está cada clase? (Detallen el estado de las fuentes considerando dos dimensiones: la primera, atributos y métodos, y la segunda, código, documentación y comentarios)***
Para la clase GraphCalculator podemos observar que es una clase pública con un atributo que recibe un string y un grafo podemos observar igualmente su respectivo constructor y 6 métodos públicos, en cuanto al código podemos observar que se manejan bastantes strings tenemos clases que retornan un valor booleano en cuanto a la documentación no tenemos mucha información no está detallado lo que hace cada método que la clase nos ofrece pero esto le dice al desarrollador que hace cada método y para qué sirve.
Para la clase Graph podemos observar que es una clase pública que cuenta con 9 métodos privados , se manejan igualmente string tenemos dos métodos que se llaman vértices y edges que retornan un entero y un" contains" que se encarga de verificar si contiene un vértice y dos equals que retornan un booleano en cuanto a la documentación no cuenta con nada al respecto y con los comentarios cuenta con uno en el método toString() que nos da un ejemplo de lo que hace.

¿Qué diferencia hay entre el código, la documentación y los comentarios?
La documentación nos da un contexto sobre las funciones que realizarán los diferentes métodos y así mismo la clase, además, esta se ve reflejada en la documentación con la finalidad de que el cliente tenga una noción del proyecto y sus funciones, mientras que, los comentarios se dejan el código para que el desarrollador entienda lo que se quiere realizar.
Por otro lado, el código es la lógica que se escribe para que las funciones de determinado proyecto funcionen correctamente.

Ingeniería reversa [En lab02.doc GraphCalculator.asta]
MDD MODEL DRIVEN DEVELOPMENT

1. Realicen el diagrama de clases correspondiente al proyecto. (No incluyan la clase de pruebas)

2. ¿Cuáles contenedores están definidos?
Los contenedores que están definidos es el TreeMap en GraphCalculator

¿Qué diferencias hay entre el nuevo contenedor, el ArrayList y el vector [] que conocemos? Consulte el API de java.
El contenedor TreeMap utiliza un árbol rojo-negro para almacenar sus elementos. Esto significa que los elementos en un TreeMap están ordenados de manera natural (o según un comparador proporcionado) y permite operaciones rápidas para las operaciones básicas como get, put y remove. Las diferencias que hay entre ArrayList y el vector [] son que el ArrayList es que la sincronización (solo un hilo puede acceder a este) está vinculada a los vectores [] pero por el lado de el ArrayList no lo está lo que quiere decir que varios subprocesos pueden acceder a él simultáneamente por lo tanto el ArrayList es más rápido en términos de rendimiento porque los subprocesos puedes estar en varias vectores a la vez igualmente el TreeMap tiene tiempo logarítmico lo cual lo hace mas rapido que los otros dos a comparar.

3. En el nuevo contenedor, ¿Cómo adicionamos un elemento? ¿Cómo lo consultamos? ¿Cómo lo eliminamos?

Conociendo Pruebas en BlueJ [En lab02.doc *.java]
De TDD → BDD (TEST → BEHAVIOUR DRIVEN DEVELOPMENT)

1. Revisen el código de la clase GraphTest.

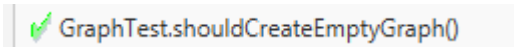
¿cuáles etiquetas tiene (componentes con símbolo@)?
Tiene 3 tipos de etiquetas, las cuales son:
@Before
@Test
@After
¿cuántos métodos tiene?
Tiene 8 métodos.

¿cuantos métodos son de prueba?
6 métodos son de prueba.

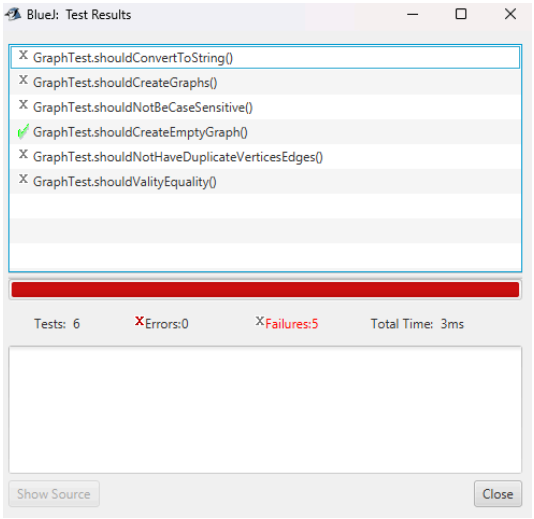
¿cómo los reconocen?
Por las respectivas etiquetas. Estas tienen la etiqueta @Test.

2. Ejecuten los tests de la clase GraphTest. (click derecho sobre la clase, Test All)
¿Cuántas pruebas se ejecutan?
Se ejecutan 6 pruebas.

¿Cuántas pasan?
Solo pasa 1.



¿Por qué?
Porque no se crea ningún grafo.
Capturen la pantalla.



3. Estudie las etiquetas encontradas en 1 (marcadas con @). Expliquen en sus palabras su significado.

@Before : Se usa para preparar el ambiente para luego realizar las diferentes pruebas.
@Test: Contiene los métodos con los diferentes casos que se van a probar.
@After: Se usa para reiniciar la prueba.

4. Estudie los métodos assertTrue, assertFalse, assertEquals, assertNull y fail de la clase Assert del API JUnit 1. Explique en sus palabras que hace cada uno de ellos.

assertTrue(boolean condition): afirma si una condición dada es verdadera.
assertTrue(String message, boolean condition) : afirma igualmente si la condición dada es verdadera.
assertFalse(boolean condition): afirma si una condición dada es falsa.
assertFalse(String message, boolean condition) : afirma igualmente si la condición dada es falsa.
assertEquals: afirma si dependiendo lo que queramos comparar si es verdadero por ejemplo dos arreglos booleanos, de tipo carácter , de tipo string , de tipo arreglos cortos de tipo flotantes , enteros … básicamente comprada si dos objetos son iguales
a***ssertNull(Object object)***: afirma que un objeto es nulo.
assertNull(String message,Object object): afirma que un objeto es nulo.
fail(): *Falla una prueba que no tenga algún mensaje.*
fail(String message) : Falla una prueba con el mensaje dado.

5. Investiguen y expliquen la diferencia entre un fallo y un error en Junit.

En Junit un fallo quiere decir que las pruebas fallan o lo que afirman son correctas y los errores ocurren cuando se producen excepciones inesperadas mientras se ejecuta la prueba.

Escriba código,usando los métodos del punto 4, para codificar los siguientes tres casos de prueba y lograr que se comporten como lo prometen shouldPass, shouldFail, shouldErr.

```
@Test
public void shouldPass(){
    assertTrue(null,10 < 20);
}
```

```
@Test
public void shouldFail(){
    assertTrue(null,9 > 10);
}
```

```
@Test
public void shouldErr(){
    String texto = null;
    int longitud = texto.length();
}
```

Practicando Pruebas en BlueJ

De TDD → BDD (TEST → BEHAVIOUR DRIVEN DEVELOPMENT)

Ahora vamos a escribir el código necesario para que las pruebas de GraphTest pasen.

1. Determinen los atributos de la clase Graph. Justifique la selección.

Los atributos de la clase Graph van a ser los vértices y arcos que son los atributos esenciales de un grafo.

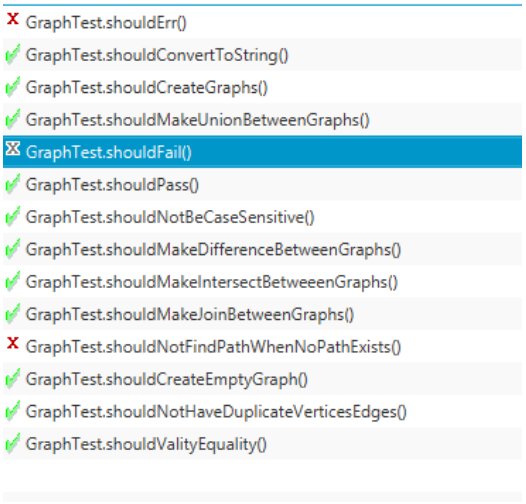
2. Determinen el invariante de la clase Graph. Justifique la decisión.

El invariante de la clase Graph es que los vértices son pueden ser iguales y así mismo sus aristas. Las aristas deben estar conectadas con exactamente dos vértices.

3. Implementen los métodos de Graph necesarios para pasar todas las pruebas definidas.¿Cuáles métodos implementaron?

Se implementaron diferentes métodos, entre ellos el equals(), deleteDuplicateVertices(), deleteDuplicateEgdes(), upperCaseVertices(), upperCaseEdges() y el toString(). Además, se crearon diferentes constructores para la realización de las operaciones.

4. Capturen los resultados de las pruebas de unidad.



4. División por Ciclos

Para la división de ciclos lo que nosotros realizamos fue dividirlo en

Ciclo 1:

Operaciones Básicas

```
public class GraphCalculator{

    private TreeMap<String,Graph> variables;
    private boolean lastOperation = false;

    public GraphCalculator(){
        variables = new TreeMap<>();
    }

    //Create a new variable
    public void create(String nombre){
        variables.put(nombre, new Graph());
    }

    //Assign a graph to an existing variable
    //a := graph
    public void assign(String graph, String[] vertices, String [][] edges ){
        if (variables.containsKey(graph)){
            variables.put(graph,new Graph(vertices,edges));
            lastOperation = true;
        }
    }

    public Graph getGraph(String graph){
        return variables.get(graph);
    }
```

Ciclo 2:

Operaciones Unarias

```
public boolean containsVertices(String vertex1,String vertex2){
    return vertices.contains(vertex1) && vertices.contains(vertex2);
}
```

```
public Graph addEdge(String vertex1,String vertex2){
    ArrayList<String> newEdges = new ArrayList<>(Arrays.asList(vertex1, vertex2));
    if (!edges.contains(newEdges)) {
        edges.add(newEdges);
    }
    return new Graph(vertices,edges);
}
```

```
public Graph removeEdge(String vertex1,String vertex2){
    ArrayList<String> newEdges = new ArrayList<>(Arrays.asList(vertex1, vertex2));
    if (edges.contains(newEdges)) {
        edges.remove(newEdges);
    }
    return new Graph(vertices,edges);
```

Ciclo 3:

Operaciones Binarias

Realizar las distintas operaciones entre grafos como por ejemplo unión, intersección, diferencia.

```
public void assignBinary(String a, String b, char op, String c){

    Graph graphB = variables.get(b);
    Graph graphC = variables.get(c);
    Graph graphA = null;
    if(op == 'u'){
        graphA = graphB.union(graphC);
        lastOperation = true;
    }
    if(op == 'i'){
        graphA = graphB.intersection(graphC);
        lastOperation = true;
    }
    if(op == 'd'){
        graphA = graphB.difference(graphC);
        lastOperation = true;
    }
    if(op == 'j'){
        graphA = graphB.join(graphC);
        lastOperation = true;
    }
    variables.put(a,graphA);
}
```

```java
@Test
public void shouldMakeUnionBetweenGraphs() {
    ArrayList<String> vertices1 = new ArrayList<>(Arrays.asList("DDYA", "MYSD", "DOPO"));
    ArrayList<ArrayList<String>> edges1 = new ArrayList<>();
    edges1.add(new ArrayList<>(Arrays.asList("DDYA", "DOPO")));
    edges1.add(new ArrayList<>(Arrays.asList("DDYA", "MYSD")));

    ArrayList<String> vertices2 = new ArrayList<>(Arrays.asList("MBDA", "POOB", "ECDI", "DOPO", "DDYA"));
    ArrayList<ArrayList<String>> edges2 = new ArrayList<>();
    edges2.add(new ArrayList<>(Arrays.asList("MBDA", "POOB")));
    edges2.add(new ArrayList<>(Arrays.asList("MBDA", "ECDI")));
    edges2.add(new ArrayList<>(Arrays.asList("DDYA", "DOPO")));
    edges2.add(new ArrayList<>(Arrays.asList("DOPO", "ECDI")));

    Graph A = new Graph(vertices1, edges1);
    Graph B = new Graph(vertices2, edges2);
    Graph unionGraph = A.union(B);

    ArrayList<String> expectedVertices = new ArrayList<>(Arrays.asList("DDYA", "MYSD", "DOPO", "MBDA", "POOB", "ECDI"));
    ArrayList<ArrayList<String>> expectedEdges = new ArrayList<>();
    expectedEdges.add(new ArrayList<>(Arrays.asList("DDYA", "DOPO")));
    expectedEdges.add(new ArrayList<>(Arrays.asList("DDYA", "MYSD")));
    expectedEdges.add(new ArrayList<>(Arrays.asList("MBDA", "POOB")));
    expectedEdges.add(new ArrayList<>(Arrays.asList("MBDA", "ECDI")));
    expectedEdges.add(new ArrayList<>(Arrays.asList("DOPO", "ECDI")));

    Graph expectedGraph = new Graph(expectedVertices, expectedEdges);
    assertEquals(expectedGraph, unionGraph);
}
```

```java
@Test
public void shouldMakeIntersectBetweeenGraphs(){
    ArrayList<String> vertices1 = new ArrayList<>(Arrays.asList("DDYA","MYSD","DOPO"));
    ArrayList<ArrayList<String>> edges1 = new ArrayList<>();
    edges1.add(new ArrayList<>(Arrays.asList("DDYA","DOPO")));
    edges1.add(new ArrayList<>(Arrays.asList("DDYA","MYSD")));

    ArrayList<String> vertices2 = new ArrayList<>(Arrays.asList("DDYA","DOPO","DORA"));
    ArrayList<ArrayList<String>> edges2 = new ArrayList<>();
    edges2.add(new ArrayList<>(Arrays.asList("DDYA","DOPO")));
    edges2.add(new ArrayList<>(Arrays.asList("DDYA","DORA")));

    Graph A = new Graph(vertices1,edges1);
    Graph B = new Graph(vertices2,edges2);
    Graph intersection = A.intersection(B);

    ArrayList<String> expectedVertices = new ArrayList<>(Arrays.asList("DDYA","DOPO"));
    ArrayList<ArrayList<String>> expectedEdges = new ArrayList<>();
    expectedEdges.add(new ArrayList<>(Arrays.asList("DDYA","DOPO")));

    Graph expectedGraph = new Graph(expectedVertices,expectedEdges);
    assertEquals(expectedGraph,intersection);
}
```

```java
@Test
public void shouldMakeDifferenceBetweenGraphs() {

    ArrayList<String> vertices1 = new ArrayList<>(Arrays.asList("DDYA", "MYSD", "DOPO", "MBDA"));
    ArrayList<ArrayList<String>> edges1 = new ArrayList<>();
    edges1.add(new ArrayList<>(Arrays.asList("DDYA", "MYSD")));
    edges1.add(new ArrayList<>(Arrays.asList("MYSD", "DOPO")));
    edges1.add(new ArrayList<>(Arrays.asList("DOPO", "MBDA")));

    ArrayList<String> vertices2 = new ArrayList<>(Arrays.asList("MYSD", "DOPO", "MBDA"));
    ArrayList<ArrayList<String>> edges2 = new ArrayList<>();
    edges2.add(new ArrayList<>(Arrays.asList("MYSD", "DOPO")));

    Graph A = new Graph(vertices1, edges1);
    Graph B = new Graph(vertices2, edges2);

    Graph differenceGraph = A.difference(B);

    ArrayList<String> expectedVertices = new ArrayList<>(Arrays.asList("DDYA"));
    ArrayList<ArrayList<String>> expectedEdges = new ArrayList<>();

    Graph expectedGraph = new Graph(expectedVertices, expectedEdges);
    assertEquals(expectedGraph, differenceGraph);
}
```

```java
@Test
public void shouldMakeJoinBetweenGraphs() {

    ArrayList<String> vertices1 = new ArrayList<>(Arrays.asList("A", "B"));
    ArrayList<ArrayList<String>> edges1 = new ArrayList<>();
    edges1.add(new ArrayList<>(Arrays.asList("A", "B")));

    ArrayList<String> vertices2 = new ArrayList<>(Arrays.asList("C", "D"));
    ArrayList<ArrayList<String>> edges2 = new ArrayList<>();
    edges2.add(new ArrayList<>(Arrays.asList("C", "D")));

    Graph A = new Graph(vertices1, edges1);
    Graph B = new Graph(vertices2, edges2);

    Graph joinGraph = A.join(B);

    ArrayList<String> expectedVertices = new ArrayList<>(Arrays.asList("A","B","C","D"));
    ArrayList<ArrayList<String>> expectedEdges = new ArrayList<>();
    expectedEdges.add(new ArrayList<>(Arrays.asList("A", "C")));
    expectedEdges.add(new ArrayList<>(Arrays.asList("A", "D")));
    expectedEdges.add(new ArrayList<>(Arrays.asList("B", "C")));
    expectedEdges.add(new ArrayList<>(Arrays.asList("B", "D")));

    Graph expectedGraph = new Graph(expectedVertices, expectedEdges);
    assertEquals(expectedGraph, joinGraph);
}
```

```java
/**
 * Method to do an union between two Graphs
 */
public Graph union(Graph g) {
    ArrayList<String> combinedVertices = new ArrayList<>();
    for (String vertex : vertices) {
        combinedVertices.add(vertex);
    }
    for (String vertex : g.vertices) {
        if (!combinedVertices.contains(vertex)) {
            combinedVertices.add(vertex);
        }
    }
    ArrayList<ArrayList<String>> combinedEdges = new ArrayList<>();
    for (ArrayList<String> edge : edges) {
        combinedEdges.add(edge);
    }
    for (ArrayList<String> edge : g.edges) {
        if (!combinedEdges.contains(edge)) {
            combinedEdges.add(edge);
        }
    }
    return new Graph(combinedVertices, combinedEdges);
}
```

```java
public Graph intersection(Graph g) {
    ArrayList<String> sameVertices = new ArrayList<>();
    for (String vertex : vertices) {
        if(g.contains(vertex)){
            sameVertices.add(vertex);
        }
    }
    ArrayList<ArrayList<String>> sameEdges = new ArrayList<>();
    for (ArrayList<String> edge : edges) {
        if(g.edges.contains(edge)){
            if (sameVertices.contains(edge.get(0)) && sameVertices.contains(edge.get(1))) {
            sameEdges.add(edge);
            }
        }
    }
    Graph intersectionGraph = new Graph(sameVertices,sameEdges);
    return intersectionGraph;
}
```

```java
public Graph difference(Graph g) {
    ArrayList<String> diffVertices = new ArrayList<>();
    ArrayList<ArrayList<String>> diffEdges = new ArrayList<>();
    // vértices de A que no están en B.
    for (String vertex : vertices) {
        if (!g.contains(vertex)) {
            diffVertices.add(vertex);
        }
    }
    // aristas de A que no están en B y vértices estén en diffVertices.
    for (ArrayList<String> edge : edges) {
        String vertex1 = edge.get(0);
        String vertex2 = edge.get(1);
        if (!g.edges.contains(edge) && diffVertices.contains(vertex1) && diffVertices.contains(vertex2)) {
            diffEdges.add(edge);
        }
    }
    return new Graph(diffVertices, diffEdges);
}
```

```java
public Graph join(Graph g){
    Graph unionGraph = this.union(g);
    ArrayList<ArrayList<String>> newEdges = new ArrayList<>();
    for (String vertex1 : vertices) {
        for (String vertex2 : g.vertices) {
            ArrayList<String> edge = new ArrayList<>(Arrays.asList(vertex1, vertex2));
            if (!unionGraph.edges.contains(edge)) {
                newEdges.add(edge);
            }
        }
    }
    Graph joinGraph = new Graph(unionGraph.vertices,newEdges);
    return joinGraph;
}
```

```java
public boolean containsVertices(String vertex1,String vertex2){
    return vertices.contains(vertex1) && vertices.contains(vertex2);
}
```

```java
public Graph addEdge(String vertex1,String vertex2){
    ArrayList<String> newEdges = new ArrayList<>(Arrays.asList(vertex1, vertex2));
    if (!edges.contains(newEdges)) {
        edges.add(newEdges);
    }
    return new Graph(vertices,edges);
}
```

```java
public Graph removeEdge(String vertex1,String vertex2){
    ArrayList<String> newEdges = new ArrayList<>(Arrays.asList(vertex1, vertex2))
    if (edges.contains(newEdges)) {
        edges.remove(newEdges);
    }
    return new Graph(vertices,edges);
}
```

```java
/**
 * Returns the amount of vertex in a Graph.
 */
public int vertices(){
    return vertices.size();
}
```

```java
/**
 * Returns the amount of edges in a Graph.
 */
public int edges(){
    return edges.size();
}
```

Ciclo	Graph Calculator	GraphCalculatorTest
1	create() assign()	shouldCreateAVariable() shouldNotCreateDuplicate()

2	assignUnary() addEdge(String vertex1,String vertex2) removeEdge(String vertex1,String vertex2) getGraph() containsVertices(String vertex1,String vertex2)	shouldAssignUnaryAddEdge() shouldAssignUnaryRemoveEdge() shouldAssignUnaryContainEdge() shouldAssignUnaryPath() shouldVerifyAssignAction() shouldVerifyAssignUnaryAction()
3	assignBinary() ok()	shouldAssignBinaryUnion() shouldAssignBinaryIntersection() shouldNotBinaryIntersection() shouldAssignBinaryDifference() shouldNotDifference() shouldAssignBinaryJoin() shouldVerifyAssignBinaryAction()

RETROSPECTIVA

1. ¿Cuál fue el tiempo total invertido en el laboratorio por cada uno de ustedes? (Horas/ Hombre)

El tiempo invertido por persona fue de alrededor de 13 Horas

2. ¿Cuál es el estado actual del laboratorio? ¿Por qué?

El estado actual del laboratorio está terminado pero nos gustaría mejorar más en cuanto a la manipulación más rápida y eficaz de los grafos.

3. Considerando las prácticas XP del laboratorio. ¿Cuál fue la más útil? ¿por qué?

La práctica XP más útil para desarrollar el laboratorio fue la programación en pareja ya que nos ayudó a complementarnos con ideas y llegar a conclusiones que nos podían servir para desarrollar este.

4. ¿Cuál consideran fue el mayor logro? ¿Por qué?

El mayor logro fue implementar todas las operaciones requeridas por el grafo tanto como los assign .

5. ¿Cuál consideran que fue el mayor problema técnico? ¿Qué hicieron para resolverlo?

Podríamos hablar sobre el path que fue el mayor problema técnico debido a que la implementación de una búsqueda ya fuera BFS o DFS.

6. ¿Qué hicieron bien como equipo? ¿Qué se comprometen a hacer para mejorar los resultados?

Como equipo resaltamos nuestra comunicación y la forma en cómo abordamos los problemas que iban surgiendo como el tiempo invertido en este laboratorio que fue distribuido de una forma equitativa y eficaz.

BIBLIOGRAFÍA

- java - What's the difference between failure and error in JUnit? - Stack Overflow
- Collection (Java Platform SE 8)
- ArrayList (Java Platform SE 8)
- java - treemap vs arraylist - perfomance & resources while iterating/adding/editing values - Stack Overflow
- Contenedores y componentes en Java – Java a tu alcance