

OBJETIVOS

Desarrollar competencias básicas para:

1. Desarrollar una aplicación aplicando BDD y MDD.
2. Realizar diseños (directa e inversa) utilizando una herramienta de modelado (astah)
3. Manejar pruebas de unidad usando un *framework* (*junit*)
4. Apropiar nuevas clases consultando sus especificaciones (API java)
5. Experimentar las prácticas XP : **Designing** Use CRC cards for design sessions. **Testing**. All code must have unit tests.

Conociendo el proyecto [En lab02.doc]

1. El proyecto “DecisionTreeCalculator” 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.

primero nos encontramos con el directorio desicionTreeCalculator en el cual se encuentra un archivo de astah con el mismo nombre y otra carpeta igualmente con el mismo nombre

luego dentro de esta ultima carpeta se encuentran un directorio llamado doc,archivos con extensiones “.class”, “.ctxt”, “.java”, y un archivo .bluej llamado package

al interior de doc se encuentran un directorio llamado resources, archivos con extensiones “.html”, “.txt”, “.js”, “.css” y un archivo sin extensión llamado package-list

y por ultimo al interior de la carpeta llamada resources se encuentra el archivo inherit.gif

2. Explore el proyecto en BlueJ ¿Cuántas clases tiene? ¿Cuál es la relación entre ellas?

existen 3 clases las cuales son DecisionTree,DecisionTreeCalculator y DecisionTreeTest

DecisionTreeCalculator usa la clase DecisionTree al igual que la clase DecisionTreeTest

¿Cuál es la clase principal de la aplicación? ¿Cómo la reconocen?

la clase principal es DecisionTree pues de esta se derivan las demás clases y sus metodos

¿Cuáles son las clases “diferentes”? ¿Cuál es su propósito?

la clase diferente es DecisionTreeTest pues es de color verde y dice <<unit test>> la cual su proposito es realizarle pruebas de unidad a la clase DecisionTree

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)

la clase DecisionTree no posee ningun tipo de documentacion unicamente cuenta con un comentario en uno de los metodos

la clase DecisionTreeCalculator unicamente cuenta con encabezado de la documentacion donde dice unicamente el autor y tiene varios comentarios en los metodos

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) ¿Qué diferencia hay entre el código, la documentación y los comentarios?

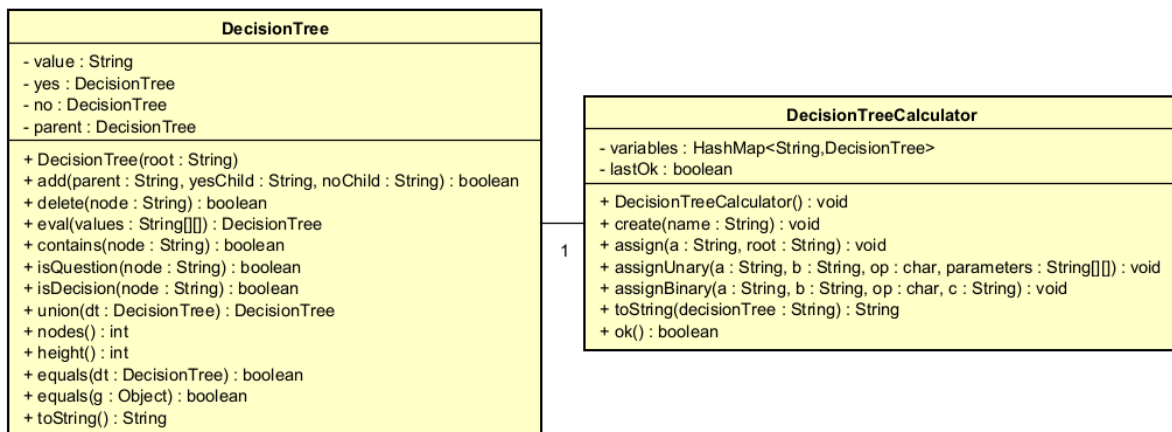
en el caso de la clase DecisionTree en la primera dimension no posee atributos y posee 13 metodos, en la segunda dimension posee el codigo de los metodos y unicamente posee un comentario en el ultimo metodo

en el caso de la clase DecisionTreeCalculator en la primera dimension posee unicamente el atributo privado de hashMap y posee 7 metodos, en la segunda dimension posee el codigo de los metodos, unicamente el encabezado de la documentacion con el nombre del autor y posee comentarios en todos los metodos a excepción del constructor

Ingeniería reversa

MDD MODEL DRIVEN DEVELOPMENT

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



2. ¿Cuáles contenedores están definidos? ¿Qué diferencias hay entre el nuevo contenedor, el ArrayList y el vector [] que conocemos? Consulte el API de java.

en el proyecto se encuentran definidos hashmap y vectores

Diferencias principales

◆ HashMap

Estructura: pares clave → valor.

Acceso promedio: $O(1)$ mediante la clave (usa tablas hash).

No mantiene orden de inserción (salvo LinkedHashMap) ni orden natural (salvo TreeMap).

Métodos clave:

put(K key, V value) → agrega o reemplaza.

get(Object key) → consulta valor.

remove(Object key) → elimina.

containsKey(Object key) / containsValue(Object value).

◆ ArrayList

Implementa una lista dinámica basada en arreglos.

Acceso por índice: $O(1)$.

Inserción/eliminación en posiciones intermedias: $O(n)$ porque requiere corrimientos.

Crece dinámicamente.

Métodos clave:

`add(E e)` → agrega al final.

`get(int index)` → consulta por índice.

`remove(int index)` o `remove(Object o)` → elimina.

`size()`.

- ♦ Vector

Muy similar a `ArrayList` (también basado en arreglos dinámicos).

Síncrono (sus métodos están sincronizados) → es seguro para hilos pero un poco más lento.

Ha sido en gran parte reemplazado por `ArrayList` en código moderno.

Métodos clave (idénticos a `ArrayList` en gran parte): `add`, `get`, `remove`, `size`.

Consulta en la API de Java (Java SE 17)

- ♦ HashMap

Pertenece al paquete `java.util`.

Implementa `Map<K,V>`.

No ordena las claves ni valores.

- ♦ ArrayList

Pertenece a `java.util`.

Implementa `List<E>`, `RandomAccess`, `Cloneable`, `Serializable`.

Mejora a los arreglos con tamaño dinámico.

- ♦ Vector

Pertenece a `java.util`.

Implementa `List<E>` y es sincronizado.

Considerado legacy, pero aún soportado.

HashMap: mapa que almacena pares clave-valor, no ordenado

ArrayList: Lista dinamica

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

para el nuevo contenedor hashmap se usan los siguientes comandos:

- put(key,value)
- get(key)
- remove(key)

Conociendo Pruebas en BlueJ

De TDD → BDD (TEST → BEHAVIOUR DRIVEN DEVELOPMENT)

Para poder cumplir con las prácticas XP vamos a aprender a realizar las pruebas de unidad usando las herramientas apropiadas. Para eso implementaremos algunos métodos en la clase DecisionTreeTest

1. Revisen el código de la clase DecisionTreeTest ¿cuáles etiquetas tiene (componentes con símbolo @)? ¿cuántos métodos tiene? ¿cuántos métodos son de prueba? ¿cómo los reconocen?

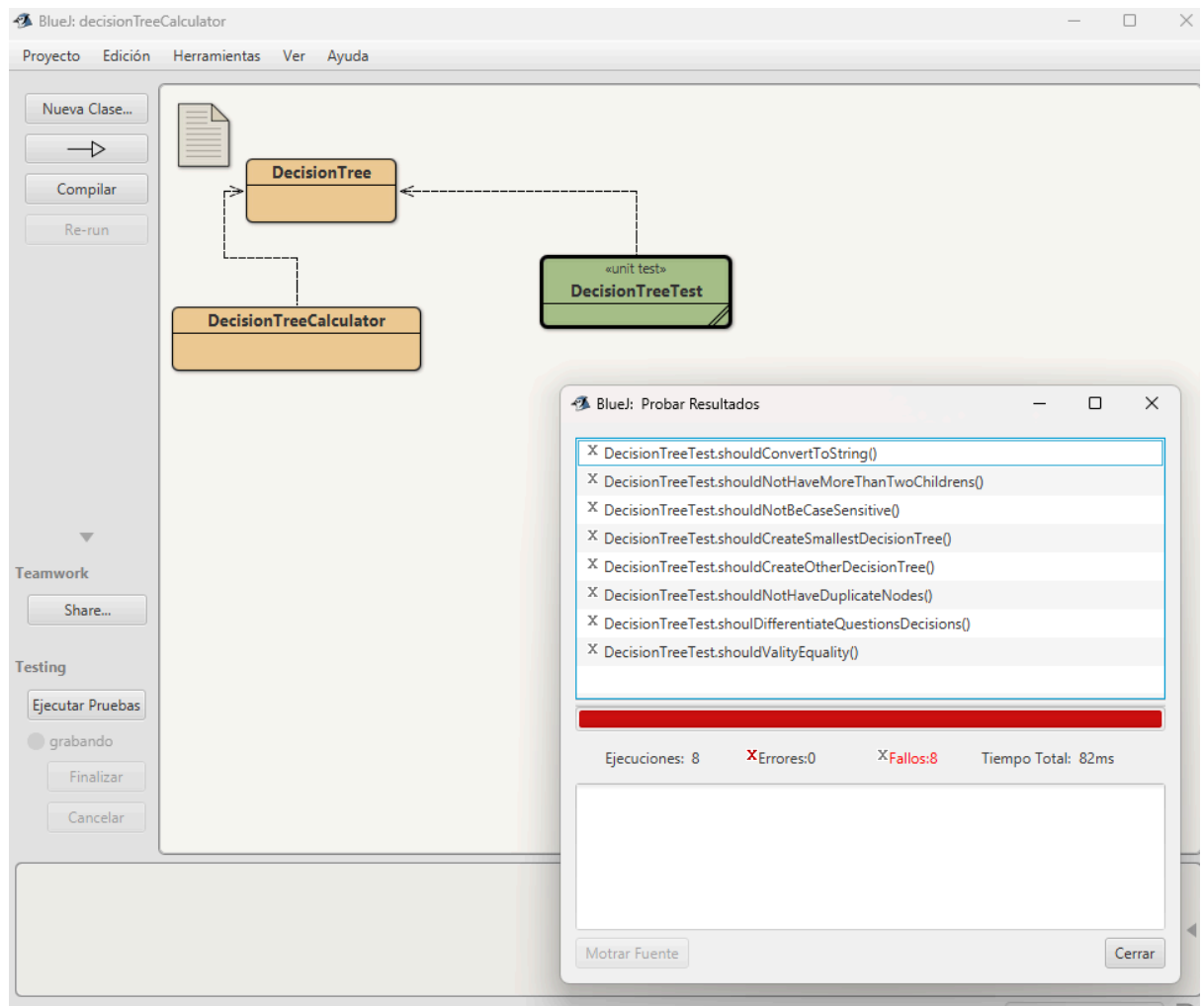
- @before
- @test
- @after

tiene 10 metodos de los cuales 8 son pruebas las cuales se reconocen por tener el

@test

2. Ejecuten los tests de la clase DecisionTreeTest. (click derecho sobre la clase, Test All) ¿cuántas pruebas se ejecutan? ¿cuántas pasan? ¿por qué? Capturen la pantalla.

se ejecutan las 8 pruebas pero ninguna pasa pues las pruebas estan diseñadas con el codigo completo y funcional en mente, por lo tanto las pruebas fallan pues se estan corriendo con la base que deja el laboratorio para que nosotros seamos quien construya esta clase para que finalmente pase las pruebas



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

- @test indica que el metodo sera una prueba unitaria
- @before hace referencia a un metodo que se va a realizar antes de cada prueba generalmente usado para reiniciar o inicializar variables
- @after hace reeferencia a un metodo que se va a ejecutar despues de cada prueba generalmente usado para liberar recurso y muchas veces evitar problemas con variables inicializadas o alteradas durante 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.

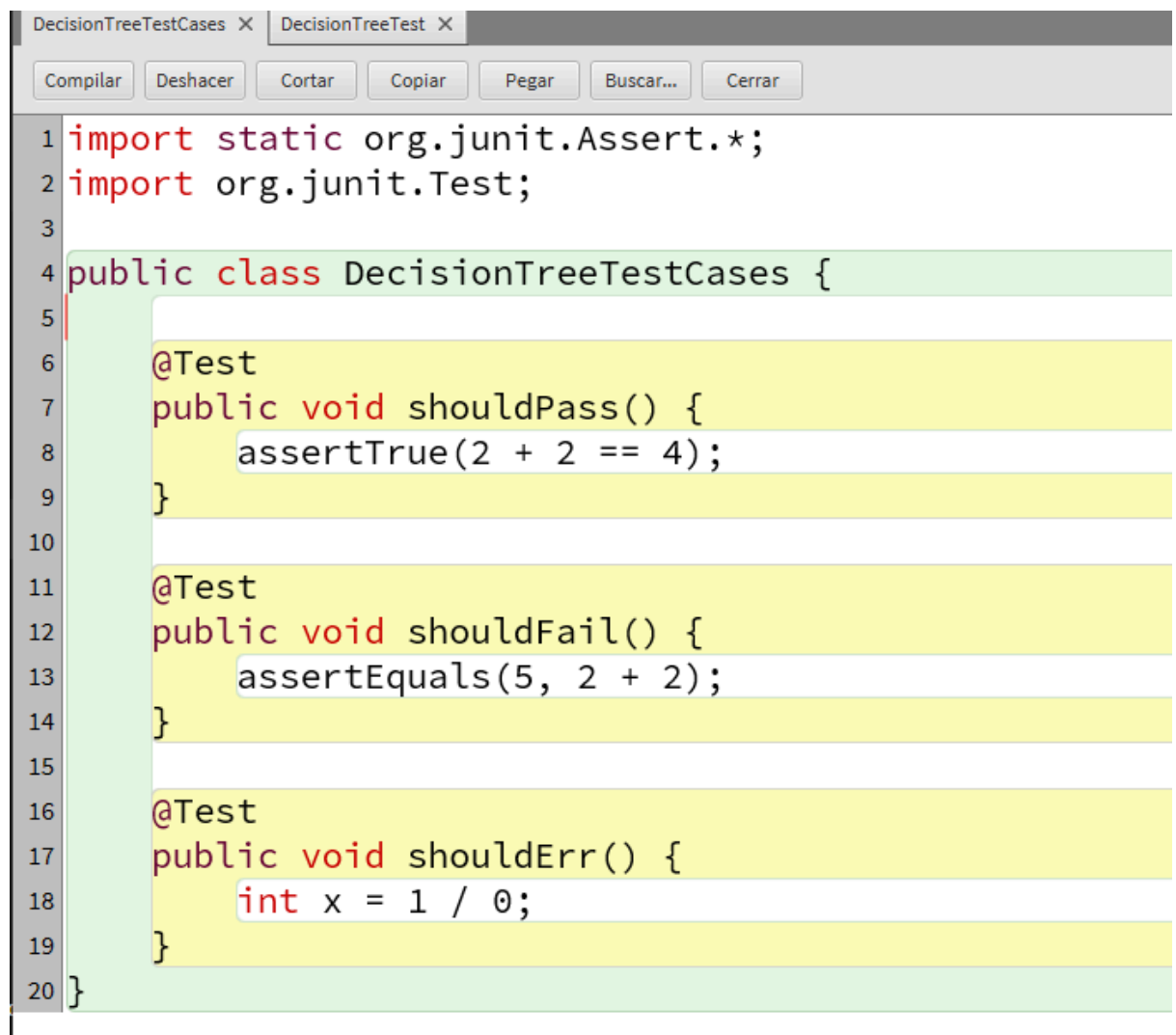
- assertTrue verifica que la condición sea verdadera y en caso de no serlo la prueba falla
- assertFalse es lo contrario a el anterior en este caso se busca que sea false la condición para que la prueba sea exitosa y de no serlo falla
- assertEquals en este caso se comparan dos objetos y si estos son iguales la prueba pasa

- assertNull verifica que el objeto sea null para que la prueba pase
- fail es una forma de hacer que la prueba falle inmediatamente por lo general usado cuando el programa no debería llegar a el punto en el que esta

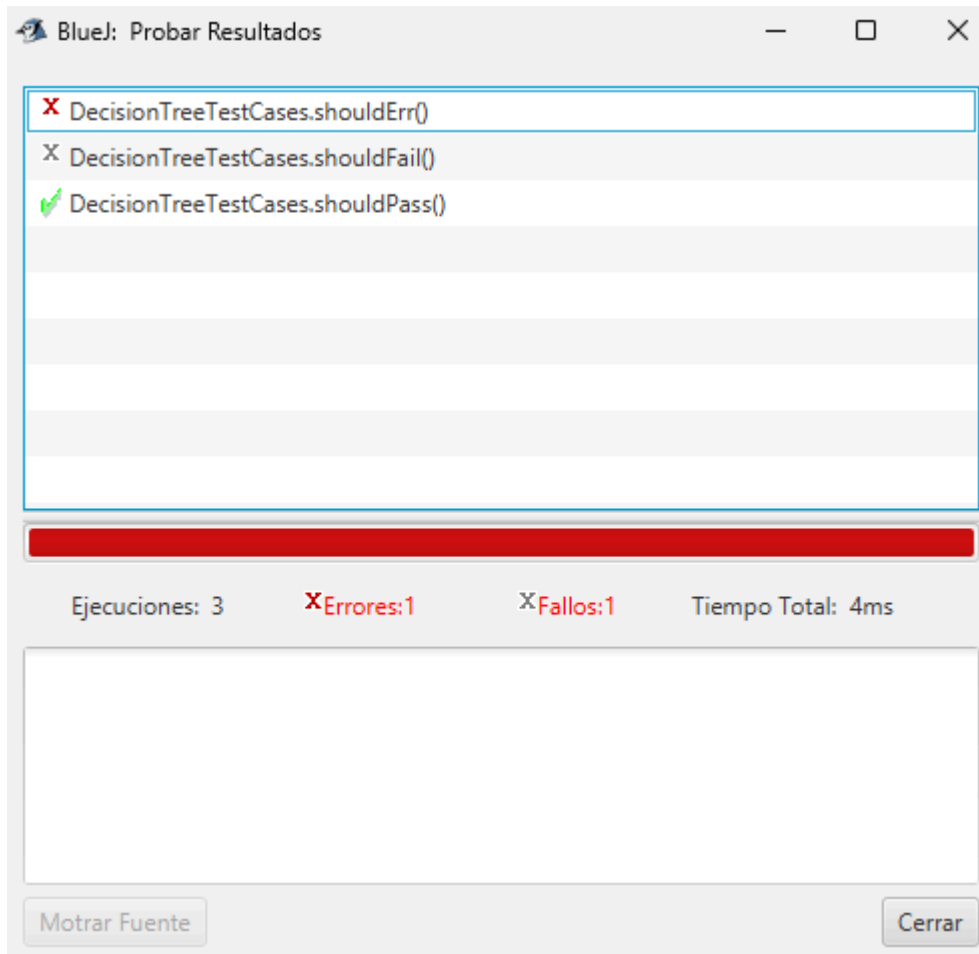
5. Investiguen y expliquen la diferencia que entre un fallo y un error en Junit. Escriba código, usando los métodos del punto 4., para codificar los siguientes tres casos de prueba y lograr que se comporten como lo prometen shouldPass, shouldFail, shouldErr.

fallo: ocurre cuando una asericion no se cumple es decir el codigo se ejecuto, pero el resultado no fue el correcto

error: ocurre cuando la prueba no se puede completar usualmente por que algo en el codigo falla usualmente produciendo excepciones



```
1 import static org.junit.Assert.*;
2 import org.junit.Test;
3
4 public class DecisionTreeTestCases {
5
6     @Test
7     public void shouldPass() {
8         assertTrue(2 + 2 == 4);
9     }
10
11     @Test
12     public void shouldFail() {
13         assertEquals(5, 2 + 2);
14     }
15
16     @Test
17     public void shouldErr() {
18         int x = 1 / 0;
19     }
20 }
```



Practicando Pruebas en BlueJ

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

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

los atributos necesarios para que funcione la clase consiste de el value que va a ser como el mensaje que tenga cada hoja del árbol, y un nodo que va a ser como la estructura de datos que vamos a trabajar cuando hacemos árboles, y por último tendríamos la raíz del árbol que es con quien creamos el árbol

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

El invariante de la clase consiste en:

- siempre hay una raíz
- cada nodo (padre) tiene máximo dos hijos (hojas)
- los valores son únicos

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

implementamos el metodo constructor de la clase

```
public DecisionTree(String root){  
    this.root=new Node(root);  
    size=1;  
}
```

implementamos la clase node, size y la heigth. Pero en esta última creamos un método privado que nos haga esto recursivo

```
public int nodes(){  
    return size;  
}
```

```
public int height(){  
    return height(root);  
}
```

```
private int height(Node node) {  
    if (node == null) return 0;  
    return 1 + Math.max(height(node.yes), height(node.no));  
}
```

seguidamente implementamos el método add, donde primero verifiquemos que todo esté bien, y luego creamos los child

```
public boolean add(String parent, String yesChild, String noChild){  
    Node parentNode = find(root, parent.toLowerCase());  
  
    if (parentNode == null) return false;  
  
    if (parentNode.yes != null || parentNode.no != null) return false;  
  
    if (contains(yesChild) || contains(noChild)) return false;  
  
    // crear hijos  
    parentNode.yes = new Node(yesChild);  
    parentNode.no = new Node(noChild);  
    size += 2;  
  
    return true;  
}
```

pero adicional a este método tenemos que crear un método auxiliar find, para buscar valores dentro del árbol

```
private Node find(Node node, String value) {  
    if (node == null) {  
        return null;  
    }  
  
    if (node.name.equals(value)) {  
        return node;  
    }  
  
    Node left = find(node.yes, value);  
    if (left != null) return left;  
  
    return find(node.no, value);  
}
```

ya con estos métodos nos funcionan estas pruebas:

X	DecisionTreeTest.shouldConvertToString()
✓	DecisionTreeTest.shouldNotHaveMoreThanTwoChildrens()
✓	DecisionTreeTest.shouldNotBeCaseSensitive()
✓	DecisionTreeTest.shouldCreateSmallestDecisionTree()
✓	DecisionTreeTest.shouldCreateOtherDecisionTree()
X	DecisionTreeTest.shouldNotHaveDuplicateNodes()
X	DecisionTreeTest.shouldDifferentiateQuestionsDecisions()
X	DecisionTreeTest.shouldValityEquality()

implementamos luego lo que fueron los métodos contains, isQuestion e isDesicion

```

public boolean contains(String node){
    return find(root, node.toLowerCase()) != null;
}

public boolean isQuestion(String node){
    Node n = find(root, node.toLowerCase());
    if (n == null) {
        return false;
    }
    return (n.yes != null || n.no != null);
}

public boolean isDecision(String node){
    Node n = find(root, node.toLowerCase());
    if (n == null) return false;
    return (n.yes == null && n.no == null);
}

```

✗	DecisionTreeTest.shouldConvertToString()
✓	DecisionTreeTest.shouldNotHaveMoreThanTwoChildrens()
✓	DecisionTreeTest.shouldNotBeCaseSensitive()
✓	DecisionTreeTest.shouldCreateSmallestDecisionTree()
✓	DecisionTreeTest.shouldCreateOtherDecisionTree()
✓	DecisionTreeTest.shouldNotHaveDuplicateNodes()
✓	DecisionTreeTest.shoulDifferentiateQuestionsDecisions()
✗	DecisionTreeTest.shouldValityEquality()

Por último el método toString lo hacemos de manera recursiva. Evaluamos si es hoja o es pregunta. Esto por lógica es mirando si tiene hijos o no

```

public String toString() {
    return toString(root);
}

private String toString(Node node) {
    if (node == null) return "";

    // Si es hoja (decision)
    if (node.yes == null && node.no == null) {
        return "(" + node.value + ")";
    }

    // Si es pregunta (tiene hijos)
    return "(" + node.value
        + " yes " + toString(node.yes)
        + " no " + toString(node.no) + ")";
}

```

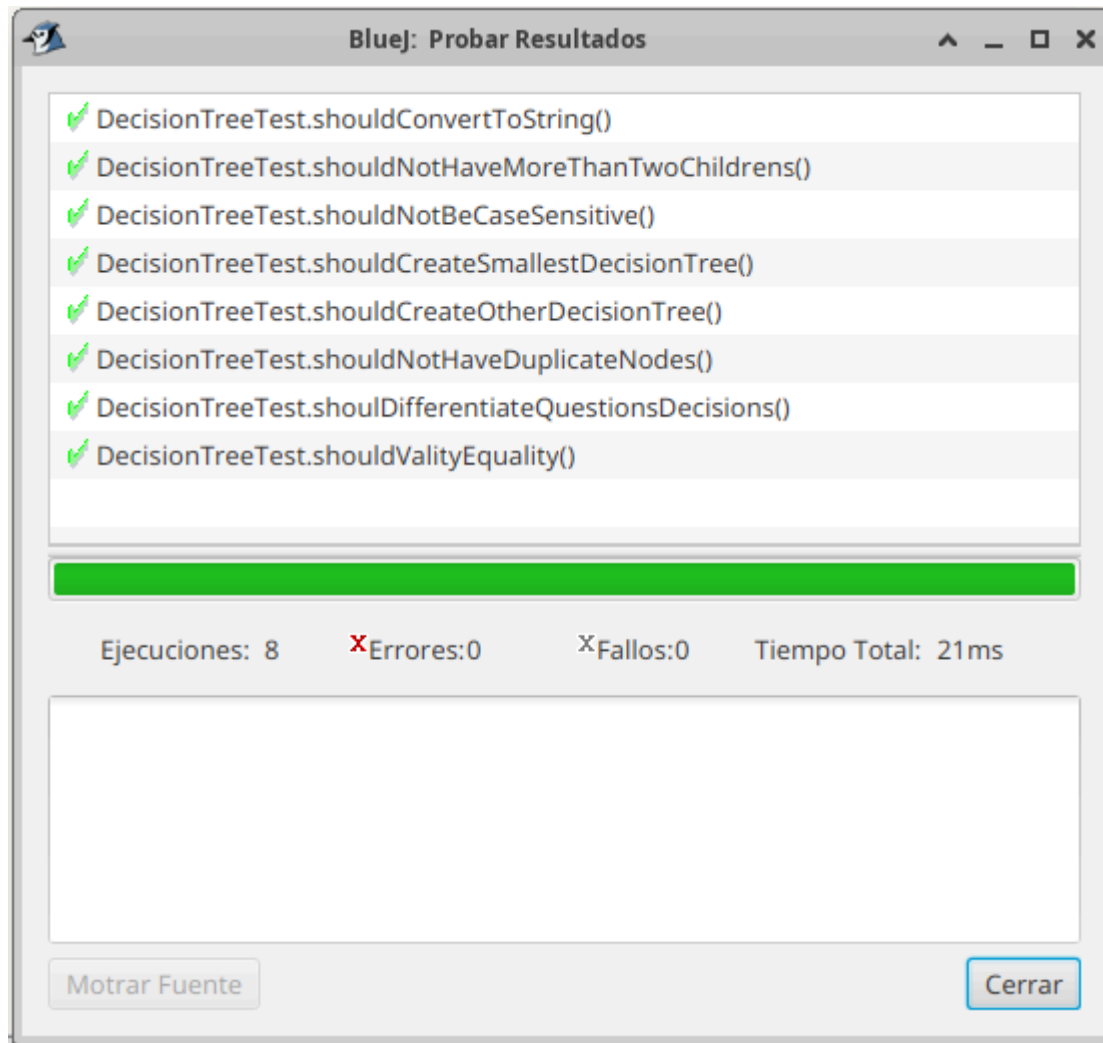
Por último trabajamos el método equal

```

public boolean equals(Object obj) {
    if (this == obj) return true;
    if (obj == null || getClass() != obj.getClass()) return false;
    DecisionTree other = (DecisionTree) obj;
    return nodesEqual(this.root, other.root);
}

```

4. Capturen los resultados de las pruebas de unidad.



Desarrollando DecisionTreeCalculator

Para desarrollar esta aplicación vamos a considerar algunos ciclos. En cada ciclo deben realizar los pasos definidos a continuación.

1. Definir los métodos base de correspondientes al mini-ciclo actual.

Primeramente los métodos base que debemos implementar son:

- `DecisionTreeCalculator` (constructor) (no secuencia)
- `create (name)`
- `assign (name, root)`
- `assignUnary`
- `assignBinary`
- `toString`
- `ok`

2. Definir y programar los casos de prueba de esos métodos Piensen en los debería y los noDeberia (should and shouldNot)

Ciclo 1 – Operaciones básicas (create, assign, toString)

- **shouldCreateAndAssignTree**: valida que se pueda crear una variable, asignarle un árbol y consultarlo con toString.

```
@Test
public void shouldCreateAndAssignTree() {
    calc.create("A");
    calc.assign("A", "Be hungry");
    assertEquals("(be hungry)", calc.toString("A"));
}
```

- **shouldNotCrashOnUnknownVariable**: valida que consultar una variable no creada retorne null y no produzca errores.

```
@Test
public void shouldNotCrashOnUnknownVariable() {
    assertNull(calc.toString("X"));
}
```

- **shouldReplaceTreeWhenReassigned**: valida que si se vuelve a asignar la misma variable, el árbol se actualiza correctamente.

```
@Test
public void shouldReplaceTreeWhenReassigned() {
    calc.create("A");
    calc.assign("A", "Be hungry");
    calc.assign("A", "Have 25");
    assertEquals("(have 25)", calc.toString("A"));
}
```

Ciclo 2 – Operaciones unarias (assignUnary)

Agregar hijos (+, usa add)

- **shouldAddChildrenWithUnaryPlus**: comprueba que se puedan agregar nodos hijos yes y no a un nodo padre existente.

```

@Test
public void shouldAddChildrenWithUnaryPlus() {
    calc.create("A");
    calc.assign("A", "Be hungry");

    String[][] params = { {"Be hungry", "Have 25", "Go to sleep"} };
    calc.assignUnary("A", "A", '+', params);

    assertEquals("(be hungry yes (have 25) no (go to sleep))", calc.toString("A"));
}

```

Evaluar árbol (?, usa eval)

- **shouldEvalTreeWithUnaryQuestion:** comprueba que al evaluar el árbol con pares [pregunta, respuesta] se llegue a la decisión correcta (por ejemplo, (buy a hamburger)).

```

@Test
public void shouldEvalTreeWithUnaryQuestion() {
    calc.create("A");
    calc.assign("A", "Be hungry");
    String[][] add1 = { {"Be hungry", "Have 25", "Go to sleep"} };
    calc.assignUnary("A", "A", '+', add1);
    String[][] add2 = { {"Have 25", "Go to restaurant", "Buy a hamburger"} };
    calc.assignUnary("A", "A", '+', add2);

    // eval: hungry = yes, have 25 = no → buy a hamburger
    String[][] answers = { {"Be hungry", "yes"}, {"Have 25", "no"} };
    calc.assignUnary("B", "A", '?', answers);

    assertEquals("(buy a hamburger)", calc.toString("B"));
}

```

Eliminar nodo (-, usa delete)

- **shouldDeleteNodeWithUnaryMinus:** valida que al eliminar un nodo, desaparece junto con sus hijos, modificando correctamente la estructura del árbol

```

@Test
public void shouldDeleteNodeWithUnaryMinus() {
    calc.create("A");
    calc.assign("A", "Be hungry");
    String[][] add1 = { {"Be hungry", "Have 25", "Go to sleep"} };
    calc.assignUnary("A", "A", '+', add1);

    // delete "Have 25"
    String[][] del = { {"Have 25"} };
    calc.assignUnary("A", "A", '-', del);

    // queda solo "Be hungry" con un hijo NO
    assertTrue(calc.toString("A").contains("go to sleep"));
}

```

Ciclo 3 – Operaciones binarias (assignBinary)

Unión (union)

- shouldUnionTwoTrees: une dos árboles simples ((x) y (y)) y comprueba que C no quede null.

```

@Test
public void shouldUnionTwoTrees() {
    calc.create("A");
    calc.assign("A", "X");
    calc.create("B");
    calc.assign("B", "Y");

    calc.assignBinary("C", "A", 'u', "B");

    assertNotNull(calc.toString("C"));
}

```

Intersección (intersection)

- shouldIntersectTwoTrees → intersecciona dos árboles con la misma raíz ((x) y (x)) y espera (x).


```

@Test
public void shouldIntersectTwoTrees() {
    calc.create("A");
    calc.assign("A", "X");
    calc.create("B");
    calc.assign("B", "X");

    calc.assignBinary("C", "A", 'i', "B");

    assertEquals("(x)", calc.toString("C"));
}

```

Diferencia (difference)

- shouldDifferenceTwoTrees → diferencia (x) con (y) y espera (x)

```

@Test
public void shouldDifferenceTwoTrees() {
    calc.create("A");
    calc.assign("A", "X");
    calc.create("B");
    calc.assign("B", "Y");

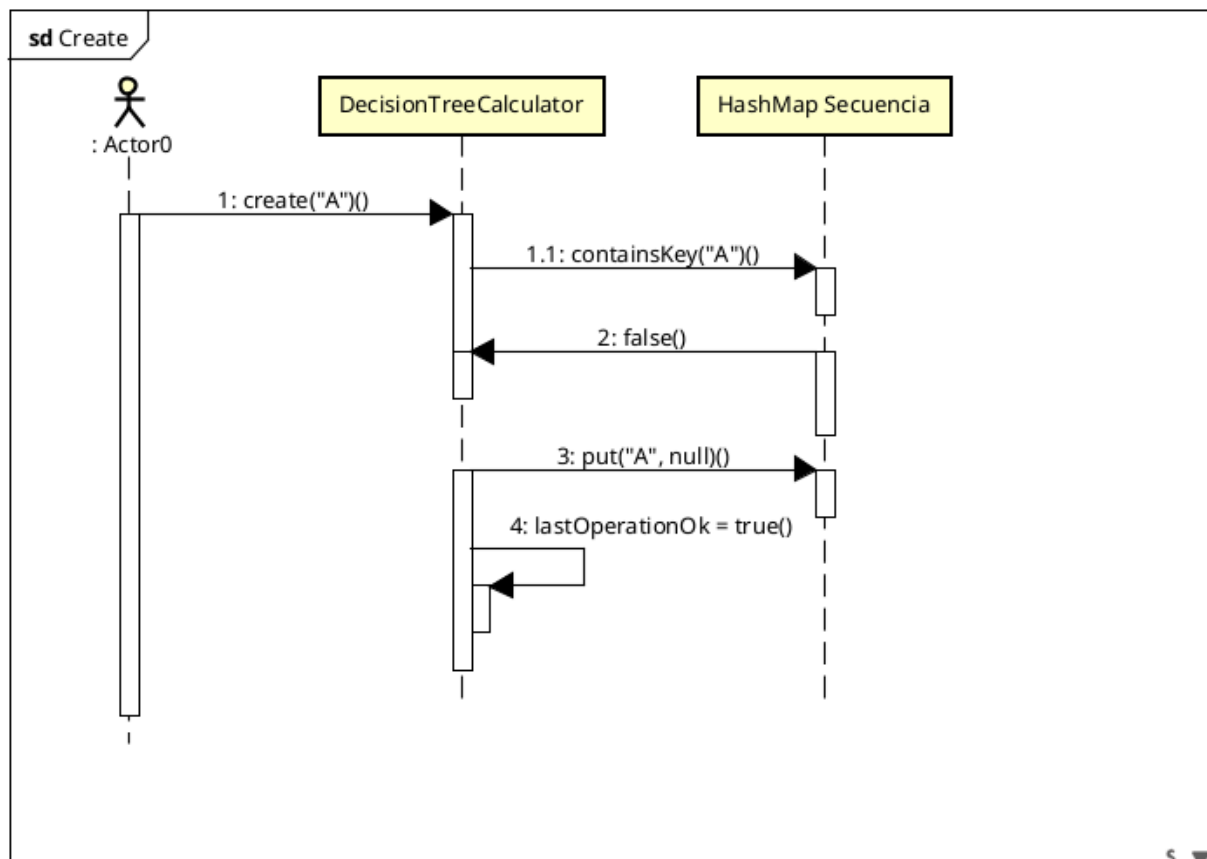
    calc.assignBinary("C", "A", 'd', "B");

    assertEquals("(x)", calc.toString("C"));
}

```

3. Diseñar los métodos Usen diagramas de secuencia. En astah, creen el diagrama sobre el método correspondiente.

4. Escribir el código correspondiente (no olvide la documentación)

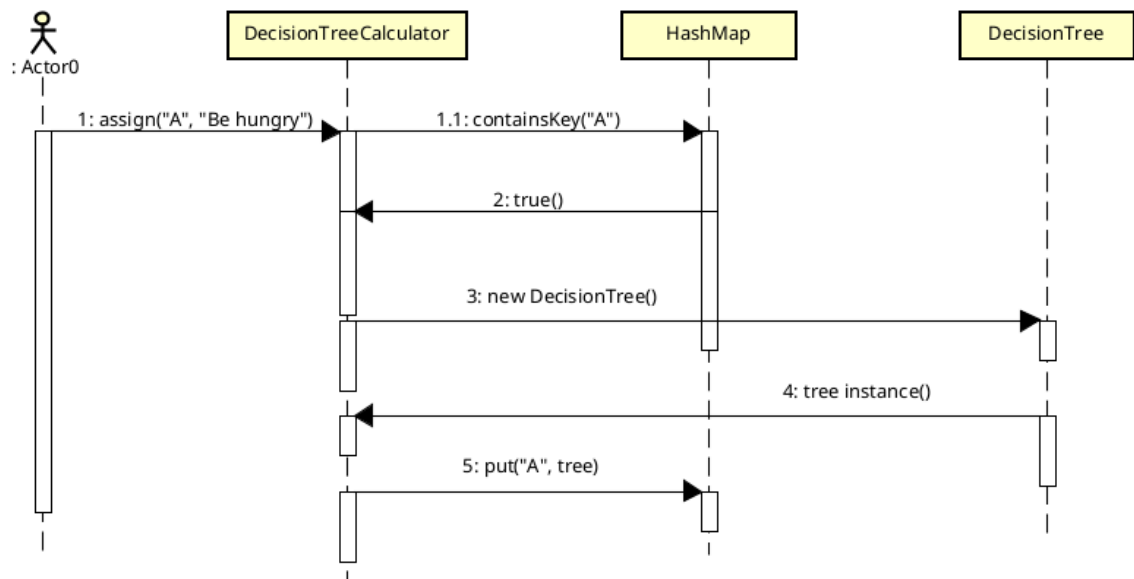


5 ▼

```
public DecisionTreeCalculator(){
    variables = new HashMap<>();
    lastOperationOk = true;
}

public void create(String name){
    if (name != null) {
        variables.put(name, null);
        lastOperationOk = true;
    } else {
        lastOperationOk = false;
    }
}
```

sd assign(String a, String root)

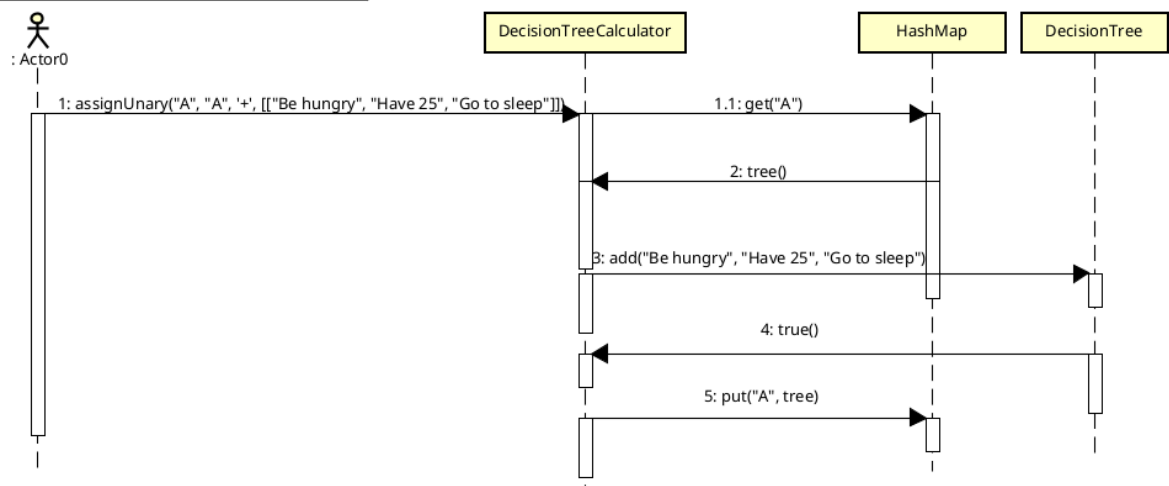


```

public void assign(String a, String root){
    if (a != null && root != null && variables.containsKey(a)) {
        variables.put(a, new DecisionTree(root));
        lastOperationOk = true;
    } else {
        lastOperationOk = false;
    }
}

```

sd assignUnary(String a, String b, '+', parameters)



```

public void assignUnary(String a, String b, char op, String[][] parameters){
    DecisionTree base = variables.get(b);
    if (base == null || a == null) {
        lastOperationOk = false;
        return;
    }

    DecisionTree result = null;

    switch(op){
        case '+':
            if (parameters != null && parameters.length > 0 && parameters[0].length == 3){
                String parent = parameters[0][0];
                String yes = parameters[0][1];
                String no = parameters[0][2];
                boolean success = base.add(parent, yes, no);
                if (success) {
                    result = base;
                    lastOperationOk = true;
                } else {
                    lastOperationOk = false;
                }
            }

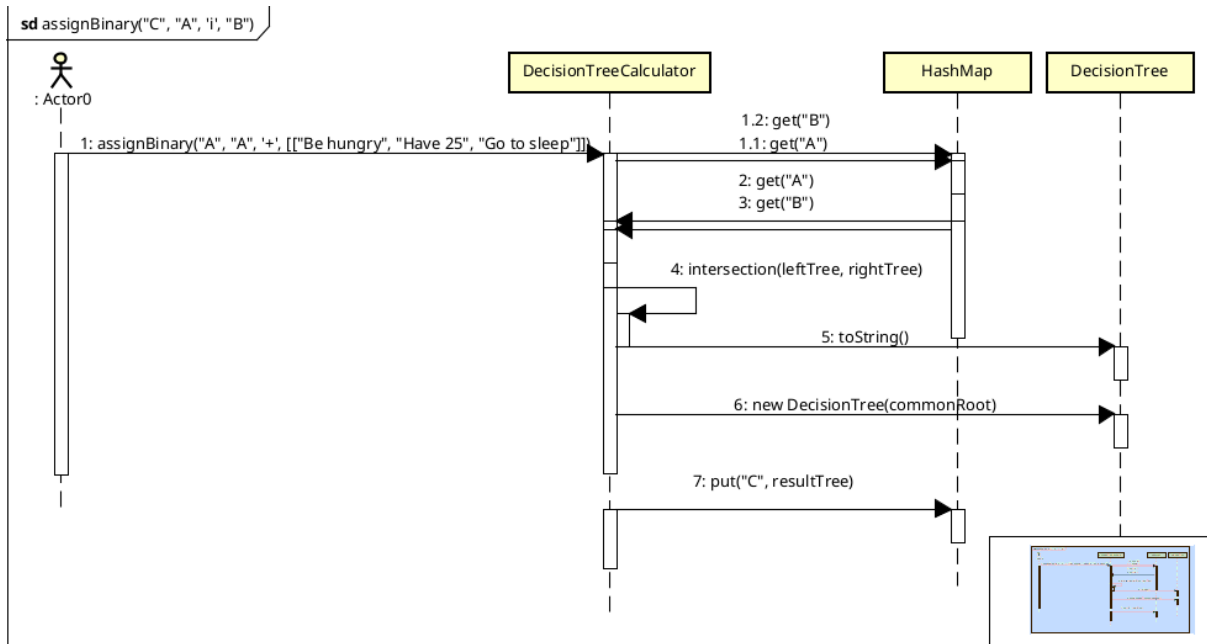
            case '-':
                if (parameters != null && parameters.length > 0 && parameters[0].length >= 1){
                    String nodeToDelete = parameters[0][0];
                    boolean success = base.delete(nodeToDelete);
                    if (success) {
                        result = base;
                        lastOperationOk = true;
                    } else {
                        lastOperationOk = false;
                    }
                } else {
                    lastOperationOk = false;
                }
            }

            case '?':
                result = base.eval(parameters);
                lastOperationOk = (result != null);
                break;

            default:
                lastOperationOk = false;
        }
    }
}

```

Nota: Aquí básicamente usamos un switch case para cada operación donde primero verifiquemos que todo esté bien, por eso se ve tan largo



```

public void assignBinary(String a, String b, char op, String c){
    DecisionTree left = variables.get(b);
    DecisionTree right = variables.get(c);

    if (left == null || right == null || a == null) {
        lastOperationOk = false;
        return;
    }

    DecisionTree result = null;

    switch(op){
        case 'u':
            result = left.union(right);
            lastOperationOk = true;
            break;

        case 'i':
            result = intersection(left, right);
            lastOperationOk = (result != null);
            break;

        case 'd':
            result = difference(left, right);
            lastOperationOk = (result != null);
            break;
    }
}

```

```

private DecisionTree intersection(D DecisionTree a, DecisionTree b){
    if (a == null || b == null) return null;

    String aStr = a.toString();
    String bStr = b.toString();

    if (aStr.equals(bStr)) {
        return new DecisionTree(a.toString().replaceAll("[()]", "").split(" ")[0]);
    }

    String aRoot = extractRoot(aStr);
    String bRoot = extractRoot(bStr);

    if (aRoot != null && aRoot.equals(bRoot)) {
        return new DecisionTree(aRoot);
    }

    return null;
}

private DecisionTree difference(D DecisionTree a, DecisionTree b){
    if (a == null) return null;
    if (b == null) return a;

    String aStr = a.toString();
    String bStr = b.toString();

    if (aStr.equals(bStr)) return null;

    String aRoot = extractRoot(aStr);
    if (aRoot != null) {
        return new DecisionTree(aRoot);
    }

    return a;
}

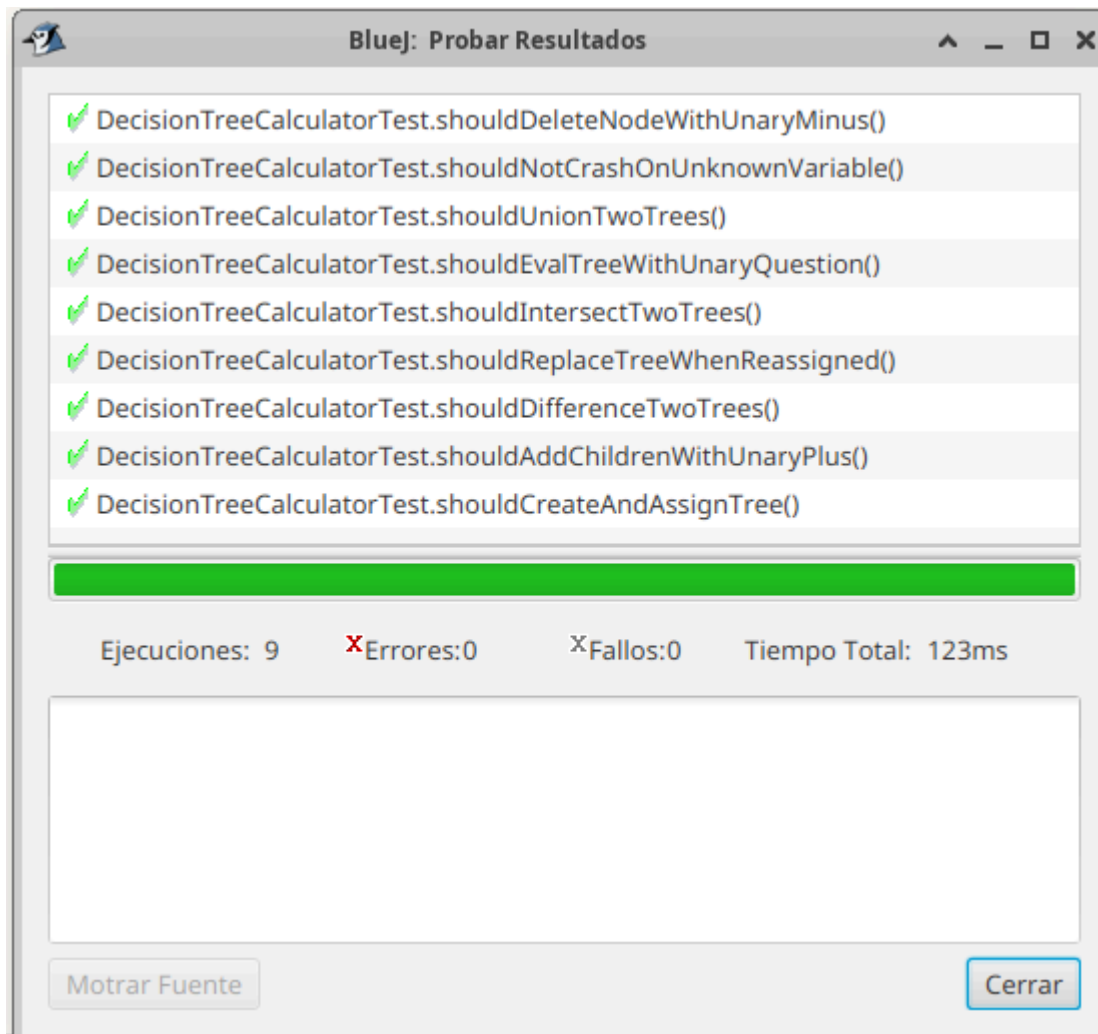
private String extractRoot(String treeStr){
    if (treeStr == null || !treeStr.startsWith("(")) return null;

    int firstSpace = treeStr.indexOf(" ");
    int firstClose = treeStr.indexOf(")");

    if (firstSpace == -1) {
        // Solo raíz: (root)
        return treeStr.substring(1, firstClose);
    } else {
        // Árbol complejo: (root yes ...)
        return treeStr.substring(1, firstSpace);
    }
}

```

5. Ejecutar las pruebas de unidad (vuelva a 3 (a veces a 2), si no están en verde)



6. Completar la tabla de clases y métodos. (Al final del documento)

Ciclo 1 : Operaciones básicas de la calculadora: crear una calculadora y asignar y consultar un árbol de decisión

Ciclo 2 : Operaciones unarias: insertar y eliminar nodos y evaluar un árbol de decisión

Ciclo 3 : Operaciones binarias: unión, intersección y diferencia.

Completen la siguiente tabla indicando el número de ciclo y los métodos asociados de cada clase.

Ciclo	DecisionTreeCalculator	DecisionTreeCalculatorTest
-------	------------------------	----------------------------

Operaciones básicas	DecisionTreeCalculator() (constructor), create(String name), assign(String a, String root), toString(String decisionTree)	shouldCreateAndAssignTree(), shouldNotCrashOnUnknownVariable(), shouldReplaceTreeWhenReassigned()
Operaciones unarias	assignUnary(String a, String b, char op, String[][] params) (casos: + insertar, - eliminar, ? evaluar)	shouldAddChildrenWithUnaryPlus(), shouldEvalTreeWithUnaryQuestion(), shouldDeleteNodeWithUnaryMinus()
Operaciones binarias	assignBinary(String a, String b, char op, String c) (casos: u unión, i intersección, d diferencia)	shouldUnionTwoTrees(), shouldIntersectTwoTrees(), shouldDifferenceTwoTrees()

RETROSPECTIVA

1. Tiempo invertido

- Daniel Ahumada: 6 horas.
- Juan Neira: 8 horas.

2. Estado actual del laboratorio

- Completado hasta el Ciclo 3 con pruebas funcionando. No se desarrolló el bono

3. Práctica XP más útil

- **TDD/BDD**: permitió implementar los métodos guiados por pruebas, asegurando que cada parte funcionara antes de avanzar.

4. Mayor logro

- Implementar y probar correctamente el DecisionTreeCalculator, pasando todos los tests de los tres ciclos.

5. Mayor problema técnico

- El método eval, que inicialmente fallaba con ciertos recorridos. Se resolvió normalizando entradas y depurando con pruebas paso a paso.

6. Trabajo en equipo

Hicimos bien la división de roles, y logramos sacar adelante el trabajo, por mejorar la organización y manejo del tiempo

7. Referencias usadas

- Documentación oficial de Java (Oracle, 2025).
- Apuntes de clase y la guía del laboratorio.
- apuntes de cursos anteriores para el tema de arboles