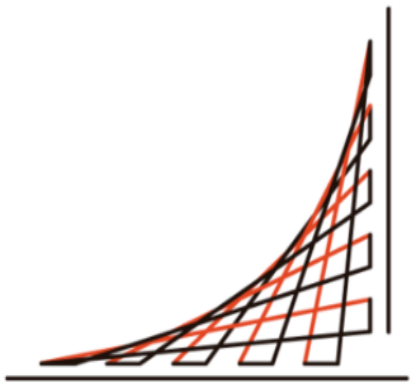


ESCUELA COLOMBIANA DE INGENIERÍA JULIO GARAVITO

UNIVERSIDAD

Nicolás Felipe Bernal Gallo

Juan Daniel Bogotá Fuetes

Desarrollo Orientada a objetos
DOPO LAB

Laboratorio #2 Diseño y pruebas Interacción entre objetos

04/09/2025

PROFESOR: María Irma Diaz Rozo

ESCUELA COLOMBIANA DE INGENIERÍA
DESARROLLO ORIENTADO POR OBJETOS [DOPO-POOB]
Diseño y Pruebas. Interacción entre objetos. 2025-2
Laboratorio 2/6

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.

ENTREGA

- ✓ Incluyan en un archivo .zip los archivos correspondientes al laboratorio. El nombre debe ser los dos apellidos de los miembros del equipo ordenados alfabéticamente.
- ✓ Deben publicar el avance (al final de la sesión) y la versión definitiva (en la fecha indicada) en los espacios preparados para tal fin

CONTEXTO

Objetivo: implementar una calculadora para árboles de decisión

Un **árbol de decisión** es una herramienta visual y lógica que se utiliza para tomar decisiones estructuradas en diversos ámbitos que van desde la inteligencia artificial hasta la economía o derecho. Se representa como un diagrama en forma de árbol, donde cada **nodo intermedio** plantea una pregunta, y cada **rama** representa una posible respuesta. Al final de cada camino, se llega a un **nodo hoja**, que indica la decisión final. En nuestro caso, el árbol de decisión que implementaremos tendrá preguntas de respuestas de tipo **si** o **no**.	

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.

   En la carpeta desicionTreeCalculator hay 12 ítems:

   1. DecisionTree.class es el archivo que corre Java (JVM).
   2. DecisionTree.ctxt es el archivo que guarda la posición de las clases en BlueJ.
   3. DecisionTree.java contiene todo el código de la clase DecisionTree.
   4. DecisionTreeCaluclator.class es el archivo que corre Java (JVM).
   5. DecisionTreeCalculator.ctxt es el archivo que guarda la posición de las clases en BlueJ.
   6. DecisionTreeCalculator.java contiene todo el código de la clase DecisionTreeCalculator.
   7. DecisionTreeTest.class es el archivo que corre Java (JVM).
   8. DecisionTreeTest.ctxt es el archivo que guarda la posición de las clases en BlueJ.
   9. DecisionTreeTest.java contiene todo el código de la clase DecisionTreeTest.
   10. package es el archivo que abre que abre el proyecto en BlueJ.
   11. README.txt Este es el archivo donde se debe describir el proyecto para un público que no tiene conocimiento.
   12. Una carpeta "doc" que contiene toda la documentación de desicionTreeCalculator.zip.

2.

¿Cuántas clases tiene? ¿Cuál es la relación entre ellas?

Tiene 3 clases, DesicionTree, DesicionTreeCalculator y DesicionTreeTest.
La relación es que DesicionTreeCalculator y DesicionTreeTest es una relación de atributo con DesicionTree.

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

La clase principal es DesicionTreeCalculator porque es la que realiza la función del programa que queremos crear.
La clave para reconocerlo fue entender que es lo que se va a realizar en el laboratorio, además

¿Cuáles son las clases "diferentes"? ¿Cuál es su propósito?

La clase diferente es DesicionTreeTest, el proposito es realizar pruebas para evaluar el funcionamiento correcto de la clase DesicionTree, por ejemplo, shouldCreateSmallestDecisionTree() prueba que se debería crear el árbol de decisión más pequeño.

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)**

Después de revisar la documentación, en las clases "DecisionTree", "DecisionTreeTest" y "DecisionTreeCalculator" no se encontró documentación en las clases.

Antes de documentar DecisionTree:

Constructor Summary

Constructors

Constructor and Description
`DecisionTree(java.lang.String root)`

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method and Description
boolean	`add(java.lang.String parent, java.lang.String yesChild, java.lang.String noChild)`
boolean	`contains(java.lang.String node)`
boolean	`delete(java.lang.String node)`
boolean	`equals(DecisionTree dt)`
boolean	`equals(java.lang.Object g)`
DecisionTree	`eval(java.lang.String[][] values)`
int	`height()`
boolean	`isDecision(java.lang.String node)`
boolean	`isQuestion(java.lang.String node)`
int	`nodes()`
java.lang.String	`toString()`
DecisionTree	`union(DecisionTree dt)`

Methods inherited from class java.lang.Object

`clone, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait`

Constructor Detail

DecisionTree

```
public DecisionTree(java.lang.String root)
```

Method Detail

add

```
public boolean add(java.lang.String parent,
                   java.lang.String yesChild,
                   java.lang.String noChild)
```

contains

```
public boolean contains(java.lang.String node)
```

delete

```
public boolean delete(java.lang.String node)
```

equals

```
public boolean equals(DecisionTree dt)
```

equals

```
public boolean equals(java.lang.Object g)
```

Overrides:
`equals in class java.lang.Object`

eval

```
public DecisionTree eval(java.lang.String[][] values)
```

height

```
public int height()
```

isDecision

```
public boolean isDecision(java.lang.String node)
```

isQuestion

```
public boolean isQuestion(java.lang.String node)
```

nodes

```
public int nodes()
```

toString

```
public java.lang.String toString()
```

Overrides:
`toString in class java.lang.Object`

union

```
public DecisionTree union(DecisionTree dt)
```

Después de documentar DecisionTree:

Class DecisionTree

java.lang.Object
DecisionTree

```
public class DecisionTree
extends Object
```

Class DecisionTree Represents a Decision Tree where each node can be either a question with two possible answers: yes/no) or a final decision. Allows building, modifying, evaluating, and comparing decision trees.

Constructor Summary

Constructors

Constructor	Description
`DecisionTree(String root)`	Constructor of the class.

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method	Description
boolean	**add**(String parent, String yesChild, String noChild)	Adds children to a given parent node.
boolean	**contains**(String node)	Checks if the tree contains a specific node.
boolean	**delete**(String node)	Deletes a node from the tree.
boolean	**equals**(DecisionTree dt)	Compares if two decision trees are equal.
boolean	**equals**(Object g)	Overrides the equals method to compare with a generic object.
DecisionTree	**eval**(String[][] values)	Evaluates the decision tree with a given set of input values.
int	**height**()	Calculates the height of the tree.
boolean	**isDecision**(String node)	Determines if a node is a decision (leaf node).
boolean	**isQuestion**(String node)	Determines if a node is a question (has children).
int	**nodes**()	Returns the number of nodes in the tree.
String	**toString**()	String representation of the decision tree.
DecisionTree	**union**(DecisionTree dt)	Joins two decision trees into one.

Methods inherited from class java.lang.Object

clone, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Details

DecisionTree [Show source in BlueJ]

```
public DecisionTree(String root)
```

Constructor of the class.

Parameters:

`root` - Root node of the decision tree.

Method Details

add [Show source in BlueJ]

```
public boolean add(String parent,
                   String yesChild,
                   String noChild)
```

Adds children to a given parent node.

Parameters:

`parent` - Parent node where children will be added.

`yesChild` - Child node corresponding to the "yes" answer.

`noChild` - Child node corresponding to the "no" answer.

Returns:

true if the insertion was successful, false otherwise.

delete [Show source in BlueJ]

```
public boolean delete(String node)
```

Deletes a node from the tree.

Parameters:

`node` - Node to be deleted.

Returns:

true if the deletion was successful, false otherwise.

eval [Show source in BlueJ]

```
public DecisionTree eval(String[][] values)
```

Evaluates the decision tree with a given set of input values.

Parameters:
`values` - Matrix with (question, answer) pairs.

Returns:
A subtree resulting from the evaluation.

contains [Show source in BlueJ]

```
public boolean contains(String node)
```

Checks if the tree contains a specific node.

Parameters:
`node` - Node to search.

Returns:
true if the node exists, false otherwise.

isQuestion [Show source in BlueJ]

```
public boolean isQuestion(String node)
```

Determines if a node is a question (has children).

Parameters:
`node` - Node to check.

Returns:
true if the node is a question, false otherwise.

isDecision [Show source in BlueJ]

```
public boolean isDecision(String node)
```

Determines if a node is a decision (leaf node).

Parameters:
`node` - Node to check.

Returns:
true if the node is a decision, false otherwise.

union [Show source in BlueJ]

```
public DecisionTree union(DecisionTree dt)
```

Joins two decision trees into one.

Parameters:
`dt` - Another decision tree.

Returns:
Resulting tree from the union.

nodes [Show source in BlueJ]

```
public int nodes()
```

Returns the number of nodes in the tree.

Returns:
Total number of nodes.

height [Show source in BlueJ]

```
public int height()
```

Calculates the height of the tree.

Returns:
Height of the tree.

equals [Show source in BlueJ]

```
public boolean equals(DecisionTree dt)
```

Compares if two decision trees are equal.

Parameters:
`dt` - Decision tree to compare.

Returns:
true if both trees are equivalent, false otherwise.

equals [Show source in BlueJ]

```
public boolean equals(Object g)
```

Overrides the equals method to compare with a generic object.

Overrides:
`equals` in class `Object`

Parameters:
`g` - Object to compare.

Returns:
true if the object is an equivalent decision tree, false otherwise.

toString [Show source in BlueJ]

```
public String toString()
```

String representation of the decision tree. Format: Trees are represented inside parentheses. Node names are in lowercase. Children must always appear in yes/no order. Example: (a yes (b yes (c) no (d)) no (e))

Overrides:
`toString` in class `Object`

Returns:
String representation of the tree.

Antes de documentar DecisionTreeTest:

Class DecisionTreeTest

java.lang.Object
  DecisionTreeTest

```
public class DecisionTreeTest
extends java.lang.Object
```

Constructor Summary

Constructors

Constructor and Description
`DecisionTreeTest()`

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method and Description
void	**setUp**() Sets up the test fixture.
void	**shouldConvertToString**()
void	**shouldCreateOtherDecisionTree**()
void	**shouldCreateSmallestDecisionTree**()
void	**shoulDifferentiateQuestionsDecisions**()
void	**shouldNotBeCaseSensitive**()
void	**shouldNotHaveDuplicateNodes**()
void	**shouldNotHaveMoreThanTwoChildrens**()
void	**shouldValityEquality**()
void	**tearDown**() Tears down the test fixture.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

DecisionTreeTest

```
public DecisionTreeTest()
```

Method Detail

setUp

```
public void setUp()
```

Sets up the test fixture. Called before every test case method.

shouldConvertToString

```
public void shouldConvertToString()
```

shouldCreateOtherDecisionTree

```
public void shouldCreateOtherDecisionTree()
```

shouldCreateSmallestDecisionTree

```
public void shouldCreateSmallestDecisionTree()
```

shoulDifferentiateQuestionsDecisions

```
public void shoulDifferentiateQuestionsDecisions()
```

shouldNotBeCaseSensitive

```
public void shouldNotBeCaseSensitive()
```

shouldNotHaveDuplicateNodes

```
public void shouldNotHaveDuplicateNodes()
```

shouldNotHaveMoreThanTwoChildrens

```
public void shouldNotHaveMoreThanTwoChildrens()
```

shouldValityEquality

```
public void shouldValityEquality()
```

tearDown

```
public void tearDown()
```

Tears down the test fixture. Called after every test case method.

Después de documentar DecisionTreeTest:

Class DecisionTreeTest

java.lang.Object
  DecisionTreeTest

```
public class DecisionTreeTest
extends Object
```

Unit tests for DecisionTree. Verifies the correct creation, modification, and comparison of decision trees under different scenarios.

Constructor Summary

Constructors

Constructor	Description
`DecisionTreeTest()`	

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method	Description
void	`setUp()`	Sets up the test fixture.
void	`shouldConvertToString()`	Tests the correct string representation of the decision tree.
void	`shouldCreateOtherDecisionTree()`	Tests the creation of a tree with multiple levels and verifies the correct number of nodes and height.
void	`shouldCreateSmallestDecisionTree()`	Tests the creation of the smallest possible decision tree (only the root).
void	`shoulDifferentiateQuestionsDecisions()`	Tests that questions and decisions are correctly differentiated.
void	`shouldNotBeCaseSensitive()`	Tests that the tree is not case-sensitive.
void	`shouldNotHaveDuplicateNodes()`	Tests that duplicate nodes are not allowed.
void	`shouldNotHaveMoreThanTwoChildrens()`	Tests that a node cannot have more than two children.
void	`shouldValityEquality()`	Tests equality between two equivalent trees.
void	`tearDown()`	Tears down the test fixture.

Methods inherited from class java.lang.Object

`clone`, `equals`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`, `wait`, `wait`

Constructor Details

DecisionTreeTest [Show source in BlueJ]

```
public DecisionTreeTest()
```

Method Details

setUp **[Show source in BlueJ]**

```
public void setUp()
```

Sets up the test fixture. Called before every test case method.

shouldCreateSmallestDecisionTree **[Show source in BlueJ]**

```
public void shouldCreateSmallestDecisionTree()
```

Tests the creation of the smallest possible decision tree (only the root).

shouldCreateOtherDecisionTree **[Show source in BlueJ]**

```
public void shouldCreateOtherDecisionTree()
```

Tests the creation of a tree with multiple levels and verifies the correct number of nodes and height.

shoulDifferentiateQuestionsDecisions **[Show source in BlueJ]**

```
public void shoulDifferentiateQuestionsDecisions()
```

Tests that questions and decisions are correctly differentiated.

shouldNotHaveDuplicateNodes **[Show source in BlueJ]**

```
public void shouldNotHaveDuplicateNodes()
```

Tests that duplicate nodes are not allowed.

shouldNotHaveMoreThanTwoChildrens **[Show source in BlueJ]**

```
public void shouldNotHaveMoreThanTwoChildrens()
```

Tests that a node cannot have more than two children.

shouldNotBeCaseSensitive **[Show source in BlueJ]**

```
public void shouldNotBeCaseSensitive()
```

Tests that the tree is not case-sensitive.

shouldConvertToString **[Show source in BlueJ]**

```
public void shouldConvertToString()
```

Tests the correct string representation of the decision tree.

shouldValityEquality **[Show source in BlueJ]**

```
public void shouldValityEquality()
```

Tests equality between two equivalent trees.

tearDown **[Show source in BlueJ]**

```
public void tearDown()
```

Tears down the test fixture. Called after every test case method.

Antes de documentar DecisionTreeCalculator:

Después de documentar DecisionTreeCalculator:

Class DecisionTreeCalculator

java.lang.Object
DecisionTreeCalculator

```
public class DecisionTreeCalculator
extends Object
```

Class DecisionTreeCalculator Manages a set of variables that store decision trees. Provides functionality to create, assign, and manipulate decision trees by applying unary and binary operations. Works as a basic interpreter to build, modify, and combine decision trees.

Author:
ESCUELA 2025-02

Constructor Summary

Constructors

Constructor	Description
`DecisionTreeCalculator()`	Constructor.

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method	Description
void	`assign(String a, String root)`	Creates a decision tree and assigns it to an existing variable.
void	`assignBinary(String a, String b, char op, String c)`	Assigns the result of a binary operation to a variable.
void	`assignUnary(String a, String b, char op, String[][] parameters)`	Assigns the result of a unary operation to a variable.
void	`create(String name)`	Creates a new variable in the system.
boolean	`ok()`	Indicates whether the last operation was successfully completed.
String	`toString(String decisionTree)`	Returns the string representation of a decision tree in alphabetical order.

Methods inherited from class java.lang.Object

clone, equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Details

DecisionTreeCalculator [Show source in BlueJ]

```
public DecisionTreeCalculator()
```

Constructor. Initializes the storage structure.

Method Details

create [Show source in BlueJ]

```
public void create(String name)
```

Creates a new variable in the system.

Parameters:
`name` - Name of the variable to be created.

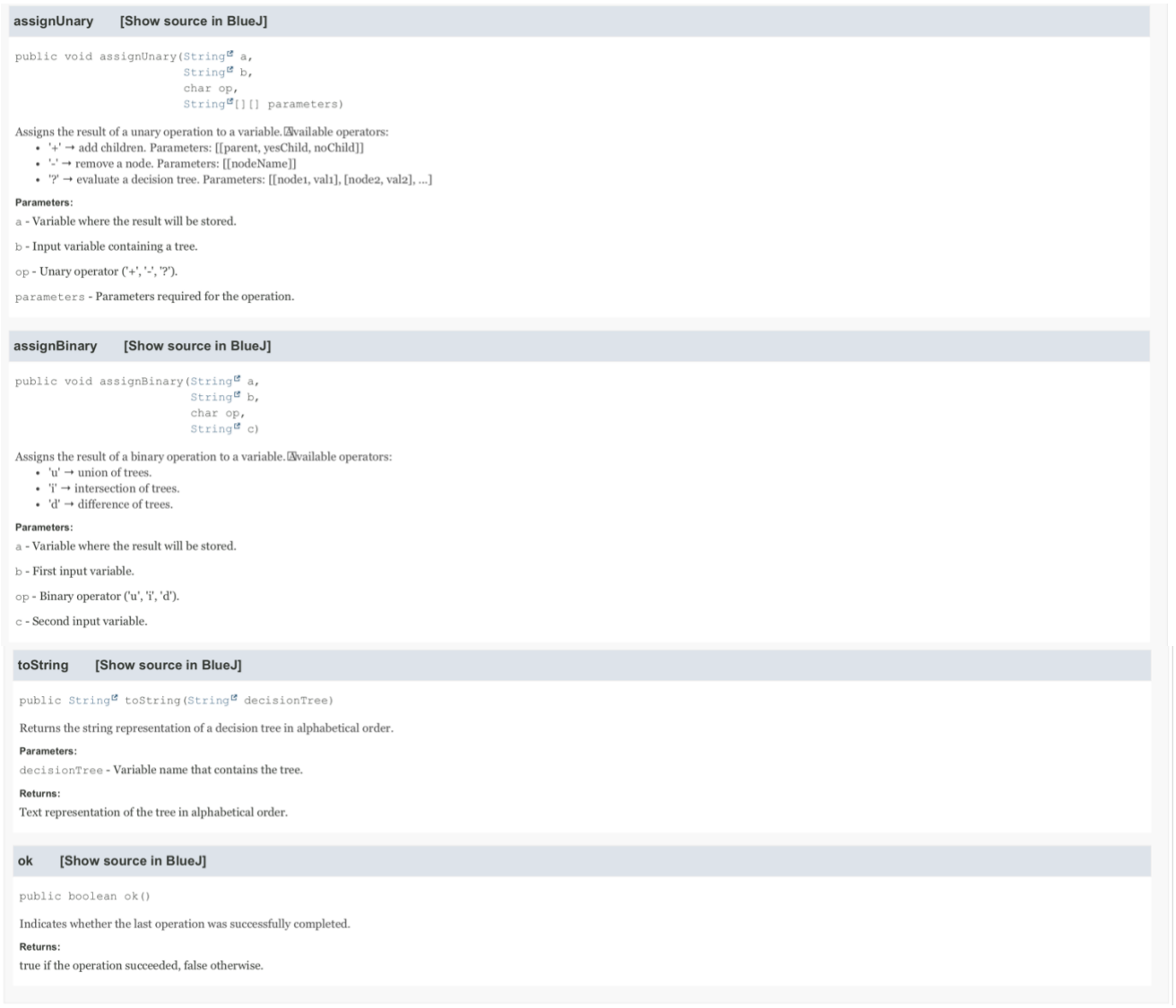
assign [Show source in BlueJ]

```
public void assign(String a,
                   String root)
```

Creates a decision tree and assigns it to an existing variable.

Parameters:
`a` - Variable name where the tree will be stored.

`root` - Root node of the decision tree.



assignUnary [Show source in BlueJ]

```
public void assignUnary(String a,
                        String b,
                        char op,
                        String[][] parameters)
```

Assigns the result of a unary operation to a variable. Available operators:
- '+' → add children. Parameters: [[parent, yesChild, noChild]]
- '-' → remove a node. Parameters: [[nodeName]]
- '?' → evaluate a decision tree. Parameters: [[node1, val1], [node2, val2], ...]

Parameters:

`a` - Variable where the result will be stored.

`b` - Input variable containing a tree.

`op` - Unary operator ('+', '-', '?').

`parameters` - Parameters required for the operation.

assignBinary [Show source in BlueJ]

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

Assigns the result of a binary operation to a variable. Available operators:
- 'u' → union of trees.
- 'i' → intersection of trees.
- 'd' → difference of trees.

Parameters:

`a` - Variable where the result will be stored.

`b` - First input variable.

`op` - Binary operator ('u', 'i', 'd').

`c` - Second input variable.

toString [Show source in BlueJ]

```
public String toString(String decisionTree)
```

Returns the string representation of a decision tree in alphabetical order.

Parameters:

`decisionTree` - Variable name that contains the tree.

Returns:

Text representation of the tree in alphabetical order.

ok [Show source in BlueJ]

```
public boolean ok()
```

Indicates whether the last operation was successfully completed.

Returns:

true if the operation succeeded, false otherwise.

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)**

DecisionTree:
No tiene atributos, tiene 13 métodos vacíos, no cuenta con documentación, tiene 2 comentarios al final del código.

DecisionTreeCalculator:
Solo tiene un atributo, llamado variables, tiene 7 métodos vacíos, no cuenta con documentación, tiene varios comentarios en todos los métodos casi que, haciendo la función de la documentación.

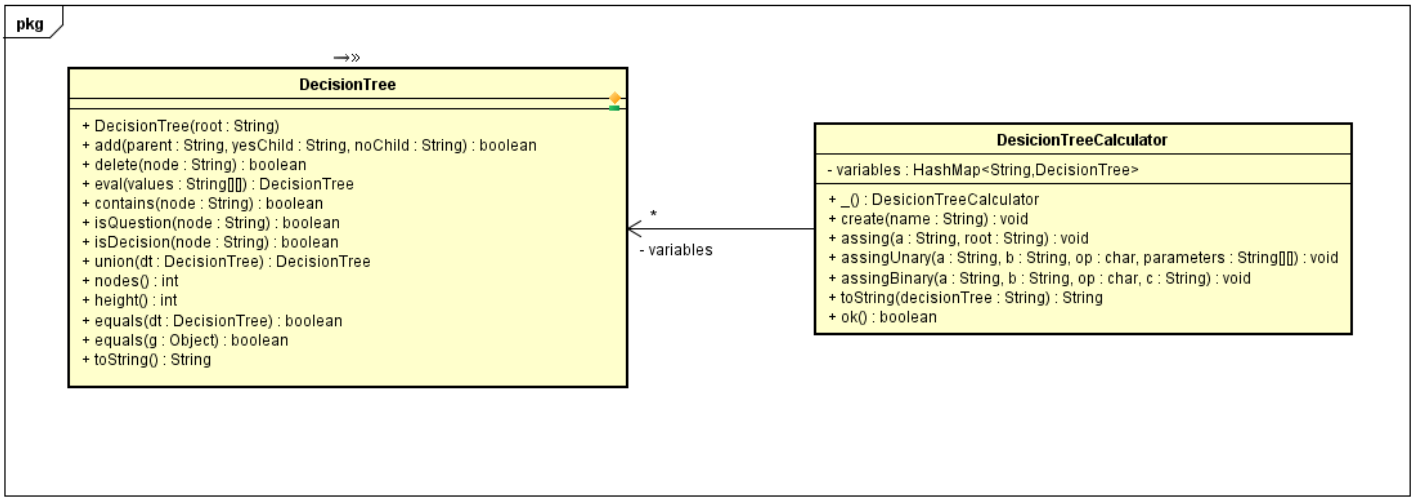
DecisionTreeTest:
No tiene atributos, tiene 10 métodos, solo tiene dos métodos con documentación, además de tener @test, @before y @after, cada método cuenta con su lógica y no tiene comentarios.

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

El comentario se usa para agregar explicaciones dentro del código, no tiene impacto en la ejecución del programa, se escribe con doble slash. La documentación se usa para describir clases y métodos de manera estructurada, se genera como formato HTML o más conocido como java.doc.

Ingeniería reversa [En lab02.doc DecisionTreeCalculator.asta]
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.

   El nuevo contenedor es “HasMap”.

HasMap	ArrayList	Vector []
Es una colección de pares, clave-valor	Lista que almacena elementos en orden de inserción.	Es una clase en java, y solo se puede modificar un hilo a la vez.
No tiene orden en los elementos.	Permite elementos duplicados.	Permite elementos duplicados.
Claves únicas.	Acceso rápido por índice.	Acceso por índice.
Permite null en valores y una única clase null.	Crece automáticamente si se excede la capacidad inicial.	Tiene tamaño fijo.
No implementa la interfaz List, pero si map.	Pertenece a la interfaz list.	No tiene métodos como add o remove.

3. **En el nuevo contenedor, ¿Cómo adicionamos un elemento?**

Para adicionar un elemento se usa el método put(K clave, V valor).

¿Cómo lo consultamos?

Se usa get(K clave) para consultar, devuelve el valor asociado a la clave.

¿Cómo lo eliminamos?

Se usa remove(K clave), que elimina el par clave–valor.

Conociendo Pruebas en BlueJ [En lab02.doc *.java]

De TDD → BDD (TEST → BEHAVIOUR DRIVEN DEVELOPMENT)

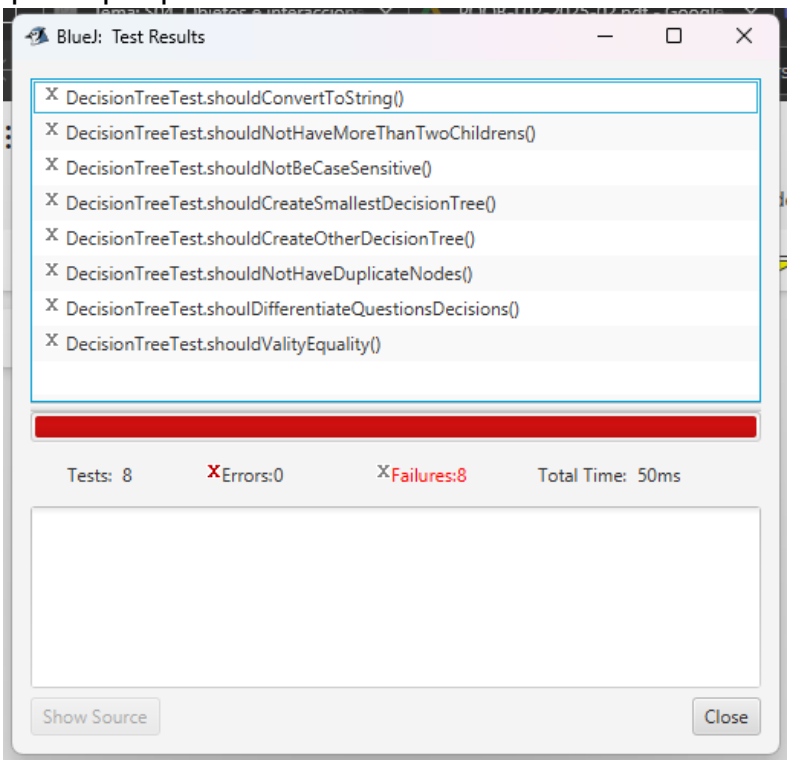
Para poder cumplir con las prácticas XP vamos a aprender a realizar las pruebas de unidad usando las herramientas apropiadas. Para eso implementaremos algunos métodos en la clase `DecisionTreeTest`

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?**

   - El método DecisionTreeTest tiene tres etiquetas @test, @before y @after.
   - Tiene 10 métodos.
   - 8 métodos son de prueba.
   - Se pueden reconocer por la etiqueta @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 8 pruebas.
   - Ninguna pasa.
   - No pasan porque la clase TERMINAR....



3. **Estudie las etiquetas encontradas en 1 (marcadas con @). Expliquen en sus palabras su significado.**
    - @Before: Indica que se ejecuta el método donde esta (en el caso de las pruebas a DecisionTree setUp()) antes de cada prueba.
    - @Test: Indica que el método es una prueba.
    - @After: Indica que se ejecuta el método donde esta (en el caso de las pruebas a DecisionTree tearDown()) después de cada 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 true, si es false falla la prueba.
    - assertFalse: verifica que la condición sea false, si es true falla la prueba.
    - assertEquals: verifica que el valor esperado sea igual al valor que se dio como resultado, si no son iguales la prueba falla.
    - assertNull: verifica que la condición sea null, si no es null falla la prueba.
    - fail: fail funciona para evitar que una prueba llegue hasta ahí en el código.

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

    En **Junit** (el framework de pruebas en Java) es muy importante diferenciar entre **fallo (failure)** y **error (error)** porque indican problemas distintos en la ejecución de las pruebas:

Fallo (Failure)

- **Definición:** Un fallo ocurre cuando una aserción (**assert**) dentro de la prueba **no se cumple**.
- **Causa:** Significa que el código **se ejecutó correctamente**, pero **el resultado no fue el esperado** según lo que definió el programador en el test.
- **Ejemplo:**

```
@Test
public void testSuma() {
    int resultado = 2 + 2;
    assertEquals(5, resultado); // Falla: 2+2 = 4, no 5
  }
```

Error (Error)

- **Definición:** Un error ocurre cuando durante la ejecución de la prueba se produce una **excepción inesperada** que interrumpe la ejecución.
- **Causa:** No es que el resultado esté mal, sino que el código **no pudo ejecutarse normalmente**. Generalmente se debe a **bugs en el código**, errores de programación o mal manejo de excepciones.
- **Ejemplo:**

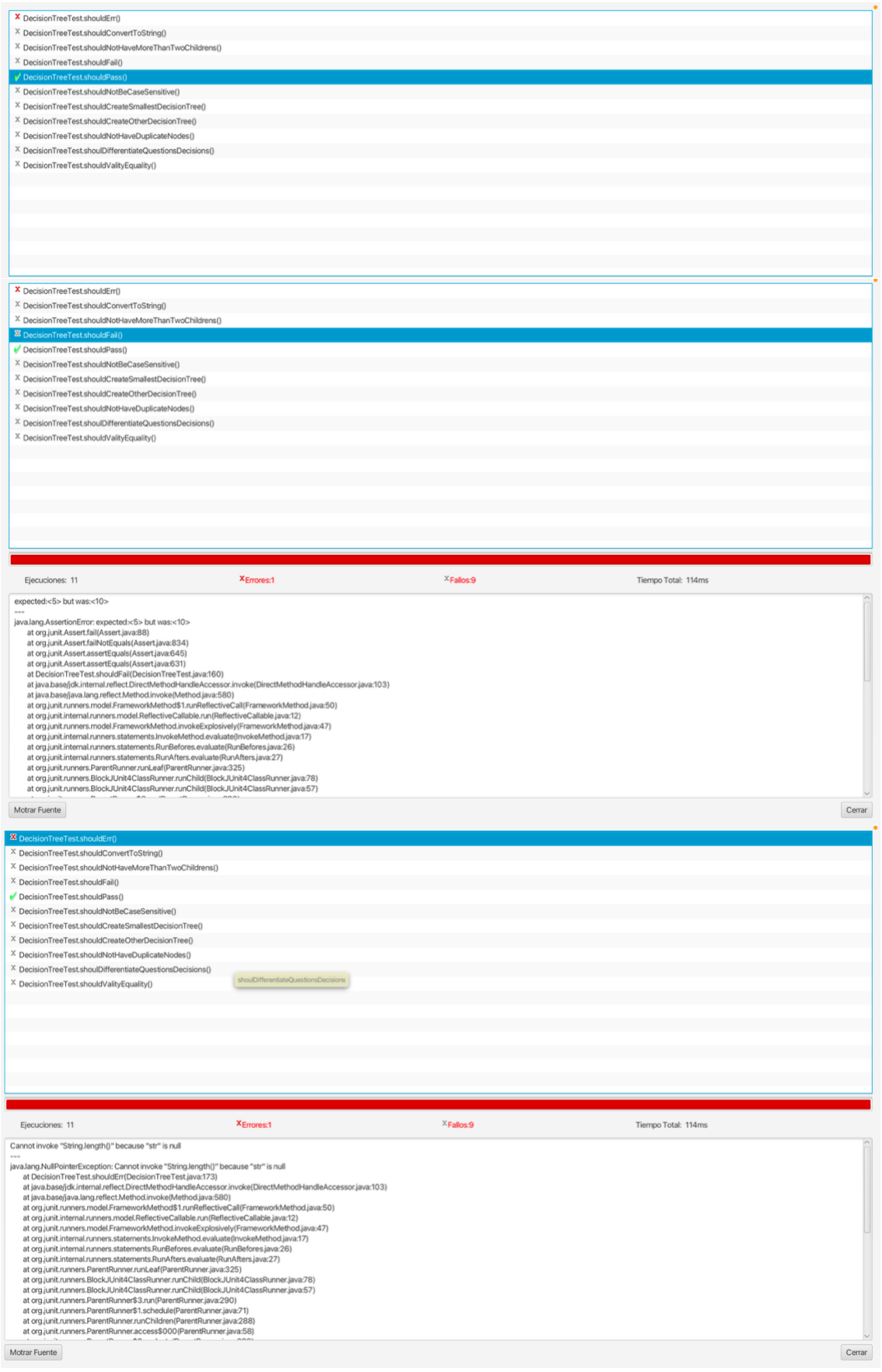
```
@Test

public void testDivision() {

int resultado = 10 / 0; // Lanza ArithmeticException

    assertEquals(2, resultado);

}
```

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`.

```java
/**
 * Test case that should pass successfully.
 *
 * This test verifies that two equal values are correctly
 * recognized as equal by the assertion.
 */
@Test
public void shouldPass() {
    int a = 5;
    int b = 5;
    // Esto es verdadero, entonces la prueba pasa
    assertEquals(a, b);
}
```

```java
/**
 * Test case that should fail.
 *
 * This test deliberately compares two different values,
 * so the assertion will fail.
 */
@Test
public void shouldFail() {
    int a = 5;
    int b = 10;
    // Esto es falso, entonces la prueba falla
    assertEquals(a, b);
}
```

```java
/**
 * Test case that should cause an error.
 *
 * This test triggers a runtime exception (NullPointerException),
 * which results in a test error.
 */
@Test
public void shouldErr() {
    String str = null;
    str.length();
}
```


DecisionTreeTest.shouldErr()
DecisionTreeTest.shouldConvertToString()
DecisionTreeTest.shouldNotHaveMoreThanTwoChildrens()
DecisionTreeTest.shouldFail()
DecisionTreeTest.shouldPass()
DecisionTreeTest.shouldNotBeCaseSensitive()
DecisionTreeTest.shouldCreateSmallestDecisionTree()
DecisionTreeTest.shouldCreateOtherDecisionTree()
DecisionTreeTest.shouldNotHaveDuplicateNodes()
DecisionTreeTest.shoulDifferentiateQuestionsDecisions()
DecisionTreeTest.shouldValityEquality()
Ejecuciones: 11
Errores:1
Fallos:9
Tiempo Total: 114ms
expected:<5> but was:<10>
java.lang.AssertionError: expected:<5> but was:<10>
at org.junit.Assert.fail(Assert.java:88)
at org.junit.Assert.failNotEquals(Assert.java:834)
at org.junit.Assert.assertEquals(Assert.java:645)
at org.junit.Assert.assertEquals(Assert.java:631)
at DecisionTreeTest.shouldFail(DecisionTreeTest.java:160)
Motrar Fuente
Cerrar
DecisionTreeTest.shouldErr()
shoulDifferentiateQuestionsDecisions
Ejecuciones: 11
Errores:1
Fallos:9
Tiempo Total: 114ms
Cannot invoke "String.length()" because "str" is null
java.lang.NullPointerException: Cannot invoke "String.length()" because "str" is null
at DecisionTreeTest.shouldErr(DecisionTreeTest.java:173)
Motrar Fuente
Cerrar

Prácticando Pruebas en BlueJ [En lab02.doc *.java]

De TDD → BDD (TEST → BEHAVIOUR DRIVEN DEVELOPMENT)

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.**

   - root, es esencial para representar la estructura del árbol.
   - yes, es un hijo para la respuesta "Sí".
   - no, es un hijo para la respuesta "No".

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

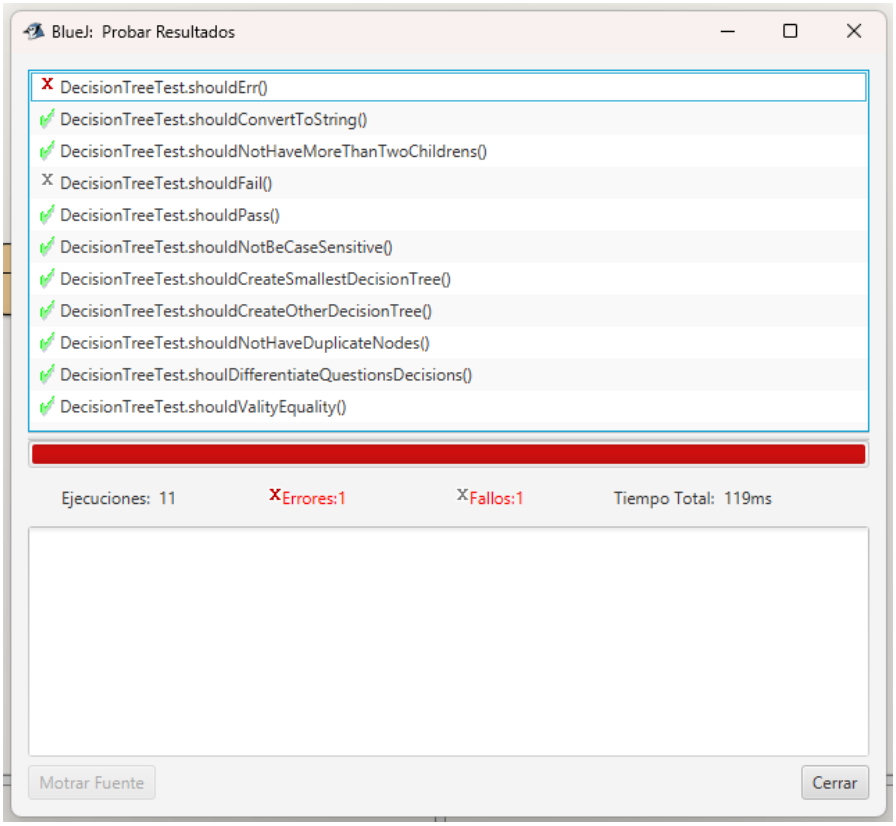
   Un invariante de clase es una condición que debe mantenerse verdadera para todos los objetos de la clase en todo momento (excepto durante la ejecución de los métodos).

   - root != null
   - root no puede ser una cadena vacía.
   - root siempre está en minúsculas.
   - Si un nodo tiene un hijo, debe tener ambos hijos (yes y no).
   - No pueden existir nodos con el mismo valor en el árbol.
   - Un nodo es pregunta si y solo si tiene dos hijos.
   - Un nodo es decisión si y solo si no tiene hijos.

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

   1. DecisionTree(String root)
   2. contains(String node)
   3. isQuestion(String node)
   4. isDecision(String node)
   5. nodes()
   6. height()
   7. equals(DecisionTree dt)
   8. toString()
   9. findNode(String node) – Método para ayudar a encontrar el nodo por el valor de este.

4. Capturen los resultados de las pruebas de unidad.



Desarrollando `DecisionTreeCalculator`

BDD - MDD

[En lab02.doc, DecisionTreeCalculator.asta, *.java]

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

Ciclo 1:

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

   - Crear : create(String name)

```
/**
    * Crea (registra) un nuevo árbol por nombre.
    * @param name nombre del árbol
    * @return true si se registró; false si name es null/"" o ya existe
    */
   public static boolean create(String name) {
      if (name == null || name.trim().isEmpty()) return false;
      if (trees.containsKey(name)) return false;
      trees.put(name, null); // creado sin raíz
      return true;
       }
```

- Asignar : assign(String a, String root)

```
/**
    * Asigna la raíz de un árbol existente.
    * @param a nombre del árbol
    * @param root etiqueta de la raíz
    * @return true si se asigna; false si el árbol no existe o root es null/""
    */
   public static boolean assign(String a, String root) {
      if (a == null || root == null || root.trim().isEmpty()) return false;
      if (!trees.containsKey(a)) return false;
      trees.put(a, root);
      return true;
       }
```

- Consultar: toString(String decisionTree)

```
/**
    * Devuelve la representación textual del árbol: "(root)".
    * @param decisionTree nombre del árbol
    * @return "(root)" si existe y tiene raíz; "" en caso contrario
    */
   public static String toString(String decisionTree) {
      if (decisionTree == null) return "";
      if (!trees.containsKey(decisionTree)) return "";
      String root = trees.get(decisionTree);
      if (root == null || root.trim().isEmpty()) return "";
      return "(" + root + ")";
       }
```

2. Definir y programar los casos de prueba de esos métodos

Piensen en los debería y los no Debería (should and shouldNot)

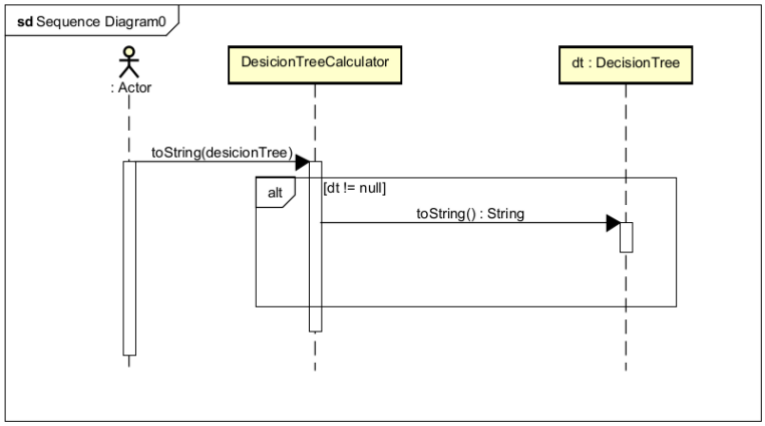
1. **shouldCreateNewDecisionTree() - Crear dos nombres de árboles**
   Verifica que se puedan registrar dos árboles distintos (arbol1, arbol2). Resultado esperado: create(...) devuelve true en ambos casos.

2. **shouldNotCreateDuplicateTree() - No crear árbol duplicado**
   Asegura que, si ya existe arbol1, no se pueda crear otro con el mismo nombre. Resultado: el segundo create("arbol1") devuelve false.

3. **shouldNotCreateTreeWithNullName() - No crear árbol con nombre nulo**
   Valida la entrada: rechaza nombres nulos o inválidos. Resultado: create(null) devuelve false.

3. Diseñar los métodos

Usen diagramas de secuencia. En astah, creen el diagrama sobre el método correspondiente.



Ciclo 2:

1. **shouldAssignRootToTree() - Asignar raíz a un árbol existente**
   Comprueba que, tras crear arbol1, se pueda asignar su nodo raíz correctamente. Resultado: assign("arbol1", "¿Tienes hambre?") devuelve true.

2. **shouldNotAssignToNonexistentTree() - No asignar raíz a árbol inexistente**
   Garantiza que no se pueda asignar una raíz a un identificador no creado. Resultado: assign("arbolInexistente", "...") devuelve false.

3. **shouldNotAssignNullRoot() - No asignar una raíz nula**
   Verifica que la raíz no puede ser nula/ vacía. Resultado: assign("arbol1", null) devuelve false.

Ciclo 3:

1. **sho1uldReturnTreeString() - Serializar el árbol a texto**
   Tras crear y asignar raíz, toString("arbol1") debe devolver la representación "(¿Tienes hambre?)". Resultado: assertEquals(...) pasa.

2. **shouldReturnEmptyForNonexistentTree() - toString de árbol inexistente**
   Confirma que, si el árbol no existe, la representación es cadena vacía. Resultado: toString("arbolInexistente") devuelve "".

3. **shouldReturnEmptyForNullTreeName() - toString con nombre nulo**
   Comprueba tolerancia a null: si el nombre es null, devuelve cadena vacía. Resultado: toString(null) devuelve "".

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

3. **Escribir el código correspondiente (no olvide la documentación)**

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

5. **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.
BONO Ciclo 4: Defina dos nuevas operaciones

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

Ciclo	DecisionTreeCalculator	DecisionTreeCalculatorTest
1	**create(String), assign(String,String), toString(String)**	**shouldCreateNewDecisionTree(), shouldNotCreateDuplicateTree(), shouldNotCreateTreeWithNullName(), shouldAssignRootToTree(), shouldNotAssignToNonexistentTree(), shouldNotAssignNullRoot(), shouldReturnTreeString(), shouldReturnEmptyForNonexistentTree(), shouldReturnEmptyForNullTreeName()**
2	**add(String,String,String,String), delete(String,String), evalToString(String,String[][]), contains(String,String), isQuestion(String,String), isDecision(String,String), nodes(String), height(String)**	**shouldAddChildrenToLeaf(), shouldRemoveLeaf(), shouldVerifyNodesExist(), shouldEvaluateFollowingAnswers()** ***(y si quieres más: shouldNotAddIfParentDoesNotExist(), shouldNotDeleteQuestion(), shouldEvalStopOnMissingAnswer())***
3	**union(String,String,String), intersection(String,String,String), difference(String,String,String), equals(String,String)**	**shouldUniteTrees(), shouldIntersectTrees(), shouldCalculateDifferenceBetweenTrees(), shouldConsiderEqualTrees()**

RETROSPECTIVA

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

   Juan Daniel Bogotá Fuentes 28 horas, más o menos 3 horas por día (jueves, lunes,martes, miércoles, jueves, viernes, sábado).
   Nicolas Felipe Bernal Gallo 28 horas (Jueves[3 horas], lunes[2 horas], martes[3 horas], miércoles[1 hora], jueves[6 horas], viernes[6 horas], sábado[7 horas])

2. **¿Cuál es el estado actual del laboratorio? ¿Por qué?**
   El estado actual del laboratorio es incompleto. Porque no pudimos realizar en su totalidad el laboratorio, profundizando en cada uno de los puntos y sus requerimientos.

3. **Considerando las prácticas XP del laboratorio. ¿cuál fue la más útil? ¿por qué?**
   Dos programadores trabajan juntos en el código (Pair Programming). Esta es la práctica XP que nos fue más útil a la hora de realizar este laboratorio, puesto que nosotros nos intercambiábamos la labor de escribir código y revisar el código de la otra persona. También porque intercambiamos ideas y conocimiento que fuimos adquiriendo en el auto estudio de cada persona.

4. **¿Cuál consideran fue el mayor logro? ¿Por qué?**
   Entender todos los requerimientos que exigía el laboratorio a partir de las preguntas que fueron expuestas para su resolución. No dejando por nuestra parte lugar a la ambigüedad de nuestras respuestas.

5. **¿Cuál consideran que fue el mayor problema técnico? ¿Qué hicieron para resolverlo?**
   El mayor problema técnico que presentamos a la hora de realizar este laboratorio fue la no interconectividad que presenta Blue J, donde no podíamos escribir en el mismo archivo al mismo tiempo, lo cual nos llevó a buscar otro entorno virtual que satisficiera esa necesidad. Luego de esto volver a Blue J y generar así el “Shapes” final.

6. **¿Qué hicieron bien como equipo? ¿Qué se comprometen a hacer para mejorar los resultados?**
   Repartir de una buena forma el trabajo para que ambos aprendamos y además tengamos la misma carga de trabajo. Nos comprometemos a repartir mejor las horas por día para hacer el laboratorio

7. **¿Qué referencias usaron? ¿Cuál fue la más útil? Incluyan citas con estándares adecuados.**

ChatGPT. (s/f). Chatgpt.com. Recuperado el 20 de febrero de 2025, de https://chatgpt.com/
Rodríguez, A. (s. f.). Análisis y casos. Grafos - software para la construcción, edición y análisis de grafos. Recuperado el 22 de febrero de 2025, de https://arodrigu.webs.upv.es/grafos/doku.php?id=analisis

Vypirailenko, A. (2023, julio 21). División entera Java. CodeGym. https://codegym.cc/es/groups/posts/es.696.division-entera-java

El mejor fue Análisis y casos, ya que una de las cosas que más nos tomó tiempo, fue conocer como debía funcionar la calculadora de grafos, esa página nos ayudó mucho con la parte teórica para su implementación.