**ESCUELA COLOMBIANA DE INGENIERÍA PROGRAMACIÓN ORIENTADA A OBJETOS**

**Diseño y Pruebas. Interacción entre objetos.**

# 2025-1 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](http://www.extremeprogramming.org/rules/crccards.html) for design sessions. **Testing** All code must have [unit tests](http://www.extremeprogramming.org/rules/unittests.html).



## 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

En matemáticas discretas, en particular en teoría de grafos, un **grafo** es una estructura que consiste en un conjunto de objetos donde algunos pares de objetos están relacionados. Los objetos son llamados vértices y cada uno de los pares de vértices relacionados se llama arco.

# 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.

* Se encuentran los directorios de graphCalculator, doc, en el primero encontramos básicamente todo para la calculadora de grafos, encontramos extensiones .class y .ctxt, además del paquete de BlueJ. En el directorio de doc, se puede encontrar la documentación de los métodos de la calculadora.

1. Exploren el proyecto en BlueJ

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

* Se tienen 3 clases: Graph, GraphCalculator, GraphTest,
  + Tenemos una relación de GraphTest con Graph, una relación de uso.
  + Tenemos Relación de GraphCalculator con Graph, una relación de atributo.

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

* La clase principal es Graph, dado que las otras dos clases usan Graph en sí para poder lograr la función de cada clase, es decir sin Graph no funcionan GraphTest ni GraphCalculator.

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

* Podemos ver que la clase de GraphTest es la “diferente”, el propósito principal de esta clase es asegurarnos que todo en nuestro código funcione correctamente.

Para las siguientes dos preguntas sólo consideren las clases “**normales**”:

1. Generen y revisen la documentación del proyecto: ¿está completa la documentación de cada clase? (Detallen el estado de documentación: encabezado y métodos)

* No, la documentación de las clases no están completas, solo aparecen los métodos y los atributos que tiene cada uno de estos, sin embargo, no aparecen las descripciones de cada uno ni el tipo de retorno.

1. 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?

* Primera dimensión: En cuanto atributos y métodos podemos ver que en el estado en el que se encuentra es relativamente avanzado, podemos encontrar los atributos y algunos métodos sin desarrollar, en la única clase donde encontramos métodos ya desarrollados es la de GraphTest.
* Segunda dimensión: En esta si encontramos una gran diferencia dado que los comentarios están mucho más completos, explicando que debería hacer cada método, mientras que en la documentación esto no está explicito.

**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)

* En GraphCalculator.asta

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

* Algunos de los contenedores que están definidos son: Array, ArrayList<E>, Vector<E>, LinkedList<E>, HashMap<K, V>, y muchos más.
* El nuevo contenedor (TreeMap) es diferente al ArrayList y al Array[] porque, en lugar de ser una lista o un arreglo, es un mapa que mantiene sus elementos ordenados por clave. Los Array[] son fijos en tamaño, y su acceso es súper rápido, pero no se pueden redimensionar. Los ArrayList son dinámicos, pero no son sincronizados, lo que puede ser un problema si estamos trabajando con múltiples hilos. El TreeMap es útil cuando necesitamos que los elementos se mantengan ordenados, pero su rendimiento no es tan rápido como un HashMap, ya que las operaciones de búsqueda, inserción y eliminación son más lentas.

Fuente:<https://www.baeldung.com/java-arraylist-vs-vector?utm_source>

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

* Adicionar: Se usa el método put().
* Consultar: Se usa el método get().
* Eliminar: Se usa el método remove().

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

## De TDD → BDD (TEST → BEHAVIOUR DRIVEN DEVELOPMENT)

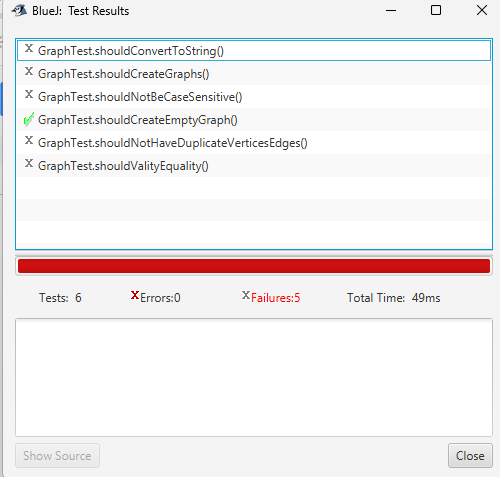
Para poder cumplir con las prácticas XP vamos a aprender a realizar las pruebas de unidad usando las herramientas apropiadas. Para eso implementaremos algunos métodos en la clase GraphTest

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

* Se tienen etiquetas de @Before, @Test, @After y se tienen 6 métodos en total. Se tiene un total de 6 métodos de prueba, esto lo sabemos por el tipo de etiqueta que en este caso es @test.

1. Ejecuten los tests de la clase GraphTest. (click derecho sobre la clase, Test All)

¿Cuántas pruebas se ejecutan? ¿Cuántas pasan? ¿por qué? Capturar la pantalla.

* Se ejecutan 6 pruebas y solo pasa 1, esto porque no hay nada construido aún, sin embargo, el de saber si el grafo está vacío si sirve. 

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

* @Before: Esta etiqueta se usa para definir un método que se ejecuta **antes** de cada prueba.
* @Test: Indica que el método marcado es una **prueba unitaria**, es decir, un test que se ejecutará para verificar que cierto comportamiento del código es correcto.
* @After: Esta etiqueta se usa para definir un método que se ejecuta **después** de cada prueba.

Referencias: <https://junit.org/junit4/>

1. Estudie los métodos assertTrue, assertFalse, assertEquals, assertNull y fail de la clase Assert del API JUnit [[1]](#footnote-1). Explique en sus palabras que hace cada uno de ellos.

* assertTrue: Verifica que la condición dada sea **true**.
* assertFalse: Verifica que la condición dada sea **falsa**.
* assertEquals: Compara dos valores y verifica que sean **iguales.**
* assertNull: Verifica que el objeto pasado sea **null.**

1. Investiguen y expliquen la diferencia 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.

* La diferencia entre un fallo y un error en Junit es que en el fallo, el test se ejecuta correctamente, sin embargo, la comparación de los resultados no da igual. Mientras que un error hace que el test no se ejecute correctamente por un problema inesperado.

# Practicando 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 GraphTest pasen.

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

**Set<String> vertices →** Un conjunto (Set) para almacenar los vértices sin duplicados.

Justificación: Se necesita evitar la duplicación de vértices y garantizar búsquedas rápidas.

**Set<Edge> edges →** Un conjunto de objetos Edge para almacenar las aristas sin duplicados.

Justificación: La prueba shouldNotHaveDuplicateVerticesEdges() sugiere que no debe haber aristas duplicadas.

**Map<String, String> normalization →** Un diccionario para manejar la insensibilidad a mayúsculas/minúsculas.

Justificación: La prueba shouldNotBeCaseSensitive() indica que "DDYA" y "Ddya" deben tratarse como el mismo nodo.

A black text on a white background

AI-generated content may be incorrect.

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

La invariante es una condición que no va a cambiar, siempre se mantiene en cualquier momento del programa, por lo que existen varias, pero mencionaremos dos:

* Los vértices y aristas deben ser únicos, sin tener duplicados
* Las aristas solo pueden conectar vértices que existen en el grafo.

1. Implementen los métodos de Graph necesarios para pasar todas las pruebas definidas.

¿Cuáles métodos implementaron?

Los métodos que se implementaron en la clase Graph para pasar las pruebas de GraphTest son:

**- Constructor Graph(String[] vertices, String[][] edges)** que se encarga de inicializar los conjuntos de vértices y aristas, asegurando que los nombres sean insensibles a mayúsculas, también valida que los vértices de las aristas existan en el grafo.

**- Método boolean contains(String vertex)** que verifica si un vértice está presente en el grafo, sin diferenciar entre mayúsculas y minúsculas.

**- Método Graph path(String start, String end)** sirve para encontrar un camino entre dos nodos en el grafo pero para las pruebas no era necesario entonces no ha sido implementado.

**- Método Graph union(Graph g)** se encarga de retornar un nuevo grafo que representa la unión del grafo actual con otro.

**-Método int vertices()** Devuelve el número total de vértices en el grafo.

**- Método int edges()** Devuelve el número total de aristas en el grafo.

**- Método boolean equals(Object obj)** Compara si dos grafos son iguales considerando sus vértices y aristas.

**- Método int hashCode()** Genera un código hash basado en los vértices y aristas.

**- Método String toString()** Devuelve una representación en cadena del grafo, ordenando las aristas alfabéticamente.

1. Capturen los resultados de las pruebas de unidad.

A screenshot of a computer

AI-generated content may be incorrect.

# Desarrollando GraphCalculator

**BDD - MDD**

[En lab02.doc, GraphCalculator.asta, \*.java]

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

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

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

**Piensen en los debería y los noDeberia (should and shouldNot)**

**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. 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 grafo

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

**-**create(string nombre)

**-**assign (String graph, String[] vertices, String[][] edges)

**-**ToString(String graph)

**-**ok()

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

**Piensen en los debería y los noDeberia (should and shouldNot)**

**“Debería”**

**-Debe crear una instancia de GraphCalculator correctamente**

**-Debe permitir crear una variable en la calculadora**

**-Debe permitir consultar la representación de un grafo existente**

**-Debe asignar un grafo a una variable existente**

**“NO Debería”**

**- NO Debería permitir asignar un grafo a una variable que no existe**

**- NO Debería permitir consultar un grafo que no existe**

**- NO Debería permitir crear una variable con un nombre vacío**

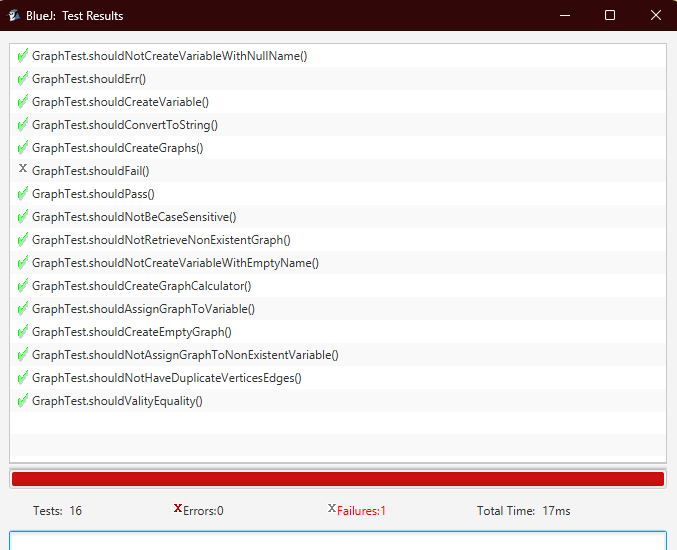
**3. Diseñar los métodos**

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

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

ok

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



Ciclo 2 : Operaciones unarias: insertar y eliminar arcos; consultar si un conjunto de vértices pertenece al graph y retornar el camino que pasa por un conjunto de vértices

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

- **Insertar un arco** (addEdge): Agregar un nuevo arco al grafo si los vértices existen.

- **Eliminar un arco** (removeEdge): Remover un arco si existe.

- **Consultar si un conjunto de vértices pertenece al grafo** (containsAllVertices).

- **Retornar un camino que pase por un conjunto de vértices** (findPathThrough).

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

**Piensen en los debería y los noDeberia (should and shouldNot)**

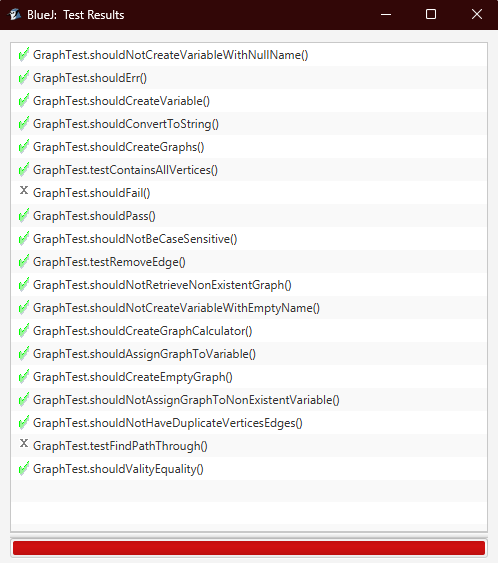
**3. Diseñar los métodos**

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

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

ok

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



Ciclo 3: Operaciones binarias: union, intersección, diferencia y junta

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

- intersection(String var1, String var2, String result)

- difference(String var1, String var2, String result)

- join(String var1, String var2, String result)

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

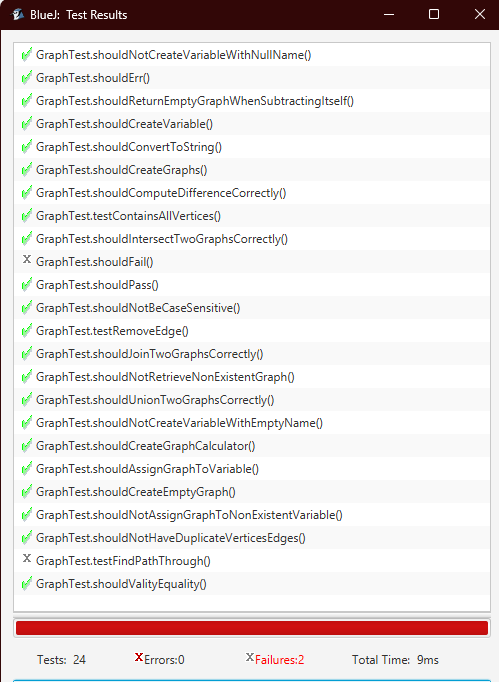
**Piensen en los debería y los noDeberia (should and shouldNot)**

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



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

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

|  |  |  |
| --- | --- | --- |
| Ciclo | GraphCalculator | GraphCalculatorTest |
| 1. | -create(string nombre)  -assign (String graph, String[] vertices, String[][] edges)  -ToString(String graph)  -ok() | shouldCreateGraphCalculator()  shouldCreateVariable()  shouldAssignGraphToVariable()  shouldNotAssignGraphToNonExistentVariable()  shouldNotRetrieveNonExistentGraph()  shouldNotCreateVariableWithEmptyName()  shouldNotCreateVariableWithNullName() |
| 2. | -addEdge(String graphName, String from, String to)  - removeEdge(String graphName, String from, String to)  -containsAllVertices(String graphName, String[] verticesToCheck)  -findPathThrough(String graphName, String[] verticesToPass) | testRemoveEdge()  testContainsAllVertices()  testFindPathThrough() |
| 3. | - intersection(String var1, String var2, String result)  - difference(String var1, String var2, String result)  - join(String var1, String var2, String result) | shouldUnionTwoGraphsCorrectly()  shouldIntersectTwoGraphsCorrectly()  shouldComputeDifferenceCorrectly()  shouldReturnEmptyGraphWhenSubtractingItself()  shouldJoinTwoGraphsCorrectly() |

## RETROSPECTIVA

1. ¿Cuál fue el tiempo total invertido en el laboratorio por cada uno de ustedes? (Horas/Hombre)
2. ¿Cuál es el estado actual del laboratorio? ¿Por qué?
3. Considerando las prácticas XP del laboratorio. ¿Cuál fue la más útil? ¿por qué?
4. ¿Cuál consideran fue el mayor logro? ¿Por qué?
5. ¿Cuál consideran que fue el mayor problema técnico? ¿Qué hicieron para resolverlo?
6. ¿Qué hicieron bien como equipo? ¿Qué se comprometen a hacer para mejorar los resultados?
7. ¿Qué referencias usaron? ¿Cuál fue la más útil? Incluyan citas con estándares adecuados.

1. [(http://junit.org/javadoc/latest/](http://junit.org/javadoc/latest/)) [↑](#footnote-ref-1)