Desarrollo de Software

Métricas de calidad

Métricas de acoplamiento

El acoplamiento se refiere a la medida en que los módulos de un sistema de software dependen unos de otros. Un bajo acoplamiento es deseable porque reduce la complejidad y mejora la mantenibilidad del sistema. Las métricas de acoplamiento ayudan a identificar las dependencias entre los componentes del software y evalúan la calidad del diseño.

1. Acoplamiento eferente (Ce)

El acoplamiento eferente mide el número de tipos o clases que un módulo particular utiliza. Cuanto mayor sea el número, más depende el módulo de otros módulos, lo que puede complicar las pruebas y el mantenimiento. Por ejemplo, un módulo que realiza llamadas a métodos de diez otros módulos tiene un acoplamiento eferente más alto que uno que solo llama a tres.

Ce=número de clases externas utilizadas por la clase actual

Ejemplo: Si tienes una clase **A** que utiliza métodos o variables de las clases **B**, **C**, y **D**, entonces: *Ce*clase A =3

2. Acoplamiento aferente (Ca)

El acoplamiento aferente mide el número de tipos o clases que dependen de un módulo particular. Un alto acoplamiento aferente indica que el módulo es central en la aplicación y que cualquier modificación puede tener amplias repercusiones. Por ejemplo, una clase utilizada por muchos otros componentes tendrá un alto Ca.

Ca=número de clases externas que utilizan la clase actual

Ejemplo: Si las clases **X**, **Y**, y **Z** utilizan la clase **A**, entonces: *Ca*clase A =3

3. Factor de acoplamiento (CF)

El factor de Acoplamiento es una métrica a nivel de sistema que evalúa el grado de acoplamiento entre los módulos de todo el sistema. Se calcula como el número de enlaces entre módulos dividido por el número máximo de enlaces posibles. Un valor más bajo de CF es generalmente mejor, indicando menor dependencia entre los módulos.

CF = e/a *(n-1)

Donde *E* es el número total de conexiones de acoplamiento reales entre módulos, *a* es el número de módulos en el sistema, y *n* es el número total de módulos.

Ejemplo de un sistema en Java con tres módulos

Supongamos que estamos construyendo un sistema de manejo de contactos, que incluye módulos para usuarios, grupos de contactos y operaciones de contacto. Aquí está una representación muy simplificada:

```
import java.util.*;

// Módulo de Usuarios

class UsuarioModulo {

    private List<Usuario> usuarios = new ArrayList<>();

    public void agregarUsuario(Usuario usuario) {

        usuarios.add(usuario);

        GrupoContactoModulo.agregarUsuarioAGrupo(usuario, "General");

    }



    public void eliminarUsuario(String nombre) {

        usuarios.removeIf(u -> u.getNombre().equals(nombre));

    }

}



// Módulo de Grupos de Contactos

class GrupoContactoModulo {

    static Map<String, List<Usuario>> grupos = new HashMap<>();
```

```
    static {
        grupos.put("General", new ArrayList<>());
    }

    public static void agregarUsuarioAGrupo(Usuario usuario, String grupoNombre) {
        grupos.get(grupoNombre).add(usuario);
    }

    public static void crearGrupo(String nombre) {
        if (!grupos.containsKey(nombre)) {
            grupos.put(nombre, new ArrayList<>());
        }
    }
}
// Módulo de Operaciones de Contacto
class ContactoOperacionesModulo {
    public void enviarMensaje(String mensaje, Usuario usuario) {
        System.out.println("Enviando mensaje a " + usuario.getNombre() + ": " + mensaje);
    }
}
```

```
// Clase Usuario
class Usuario {
    private String nombre;
    public Usuario(String nombre) {
        this.nombre = nombre;
    }
    public String getNombre() {
        return nombre;
    }
}

// Clase Principal para ejecutar el sistema
public class SistemaContactos {
    public static void main(String[] args) {
        UsuarioModulo usuarioModulo = new UsuarioModulo();
        Usuario nuevoUsuario = new Usuario("Juan");
        usuarioModulo.agregarUsuario(nuevoUsuario);

        ContactoOperacionesModulo operacionesModulo = new ContactoOperacionesModulo();
        operacionesModulo.enviarMensaje("¡Hola!", nuevoUsuario);
```

```
        GrupoContactoModulo.crearGrupo("Amigos");

        GrupoContactoModulo.agregarUsuarioAGrupo(nuevoUsuario, "Amigos");

    }

}
```

Ejercicio: Análiza el Acoplamiento y realiza el cálculo del factor de acoplamiento (CF)

4. Instability (I)

La inestabilidad de un módulo se mide utilizando la fórmula I = $Ce/Ca + Ce$. Este índice varía de 0 a 1, donde 0 indica un módulo completamente estable (sin dependencias eferentes) y 1 un módulo completamente inestable (sin dependencias aferentes). Este valor ayuda a entender la volatilidad potencial de un módulo dentro de la arquitectura.

Métricas de Cohesión

La cohesión se refiere a la medida en que las tareas realizadas por un solo módulo son funcionalmente relacionadas. Una alta cohesión dentro de un módulo es deseable porque significa que el módulo realiza una tarea bien definida y no está sobrecargado con responsabilidades que no están estrechamente relacionadas.

1. Lack of Cohesion of Methods (LCOM)

Originalmente propuesto por Chidamber y Kemerer, LCOM mide la cohesión de una clase calculando la diferencia entre el número de pares de métodos que no comparten campos de clase y aquellos que sí. Un valor alto en LCOM sugiere que una clase podría ser dividida en múltiples clases más cohesivas.

$LCOM = |P| - |Q|$ Donde $|P|$ es el número de pares de métodos en la clase que no comparten atributos de instancia y $|Q|$ es el número de pares de métodos que sí comparten atributos de instancia. Si $LCOM$ resulta negativo, se considera cero, indicando alta cohesión.

Un ejemplo detallado de código Java que simula el cálculo de LCOM para una clase ficticia. Este ejemplo muestra cómo se puede comenzar a desarrollar una herramienta que analiza y calcula la métrica LCOM para una clase Java específica. El código incluirá una estructura básica de análisis de métodos y atributos para calcular $|P|$ y $|Q|$.

```
import java.util.*;
```

```
public class LCOMCalculator {
    private static class ClassInfo {
        List<String> methods = new ArrayList<>();
        Map<String, Set<String>> methodAttributes = new HashMap<>();
        Set<String> attributes = new HashSet<>();
        public void addMethod(String methodName, Set<String> attrs) {
            methods.add(methodName);
            methodAttributes.put(methodName, attrs);
            attributes.addAll(attrs);
        }
    }
    public static void main(String[] args) {
        ClassInfo classInfo = new ClassInfo();
        // Simulación de entrada de métodos y sus accesos a atributos
        classInfo.addMethod("method1", new HashSet<>(Arrays.asList("attr1", "attr2")));
        classInfo.addMethod("method2", new HashSet<>(Arrays.asList("attr2")));
        classInfo.addMethod("method3", new HashSet<>(Arrays.asList("attr3")));
        int p = 0, q = 0;
        List<String> methods = classInfo.methods;
        for (int i = 0; i < methods.size(); i++) {
```

```
            for (int j = i + 1; j < methods.size(); j++) {
                String method1 = methods.get(i);
                String method2 = methods.get(j);
                Set<String> attrs1 = classInfo.methodAttributes.get(method1);
                Set<String> attrs2 = classInfo.methodAttributes.get(method2);
                // Calculamos si comparten atributos
                Set<String> intersection = new HashSet<>(attrs1);
                intersection.retainAll(attrs2);

                if (intersection.isEmpty()) {
                    p++; // No comparten atributos
                } else {
                    q++; // Comparten al menos un atributo
                }
            }
        }
        int lcom = p - q;
        System.out.println("LCOM = " + lcom);
    }
}
```

Ejercicio 1: Extender LCOMCalculator para incluir más métodos y atributos

Objetivo: Mejorar la comprensión de cómo la estructura y la cohesión de la clase afectan la métrica LCOM.

Añadir más métodos y atributos a la clase **ClassInfo** simulada en **LCOMCalculator**.

- Añade al menos 5 métodos adicionales con diferentes combinaciones de atributos. Cada método debe tener un propósito simulado claro (p.ej., manipular o acceder a diferentes combinaciones de atributos).
- Ejecuta el programa y observar cómo cambian los valores de *p* y *q* y, por lo tanto, cómo afecta al valor de LCOM.

Ejercicio 2: Refactorizar LCOMCalculator para mejorar la legibilidad y eficiencia

Refactorizar el código de **LCOMCalculator** para mejorar la legibilidad y eficiencia.

Puntos clave para la refactorización:

- Extraer la lógica del cálculo de LCOM a un método separado.
- Mejorar la estructura de **ClassInfo** para facilitar la adición de nuevos métodos y atributos.
- Implementar manejo de errores y validaciones para la entrada de datos.

Presenta el código refactorizado que sea más limpio, más modular, y potencialmente más eficiente.

Ejercicio 3: Implementar unidades de pruebas para LCOMCalculator

Escribe pruebas unitarias para **LCOMCalculator**, especialmente para la lógica de cálculo de LCOM. Presenta un conjunto de pruebas unitarias que validen la correcta funcionalidad del calculador LCOM.

- Usar JUnit para crear pruebas unitarias que verifiquen diferentes escenarios de cálculo de LCOM. Asegurate de que las pruebas cubran casos con alta cohesión, baja cohesión, y sin métodos.

2. LCOM4

LCOM4 es una medida que cuenta cuántos conjuntos de métodos diferentes hay en una clase que son mutuamente no accesibles. Es decir, si no hay una secuencia de accesos a campos compartidos que pueda conectar un método con otro en la clase. Cada conjunto así define un componente potencialmente separable de la clase, sugiriendo que la clase puede estar realizando demasiadas funciones.

LCOM4 = número de componentes conectados

Este ejemplo creará una estructura básica para determinar los componentes conexos entre métodos basados en el acceso a atributos compartidos.

```
import java.util.*;

 public class LCOM4Calculator {

   private static class ClassInfo {

      List<String> methods = new ArrayList<>();

      Map<String, Set<String>> methodAttributes = new HashMap<>();

      Set<String> attributes = new HashSet<>();

      public void addMethod(String methodName, Set<String> attrs) {

         methods.add(methodName);

         methodAttributes.put(methodName, attrs);

         attributes.addAll(attrs);

      }



      public int calculateLCOM4() {

         Map<String, Set<String>> graph = new HashMap<>();

         for (String method : methods) {

            graph.put(method, new HashSet<>());

         }

         for (int i = 0; i < methods.size(); i++) {

            for (int j = i + 1; j < methods.size(); j++) {

               String method1 = methods.get(i);
```

```
            String method2 = methods.get(j);
            Set<String> attrs1 = methodAttributes.get(method1);
            Set<String> attrs2 = methodAttributes.get(method2);
            // Crear un arco si comparten atributos
            if (!Collections.disjoint(attrs1, attrs2)) {
                graph.get(method1).add(method2);
                graph.get(method2).add(method1);
            }
        }
    }
    return countComponents(graph);
}

private int countComponents(Map<String, Set<String>> graph) {
    Set<String> visited = new HashSet<>();
    int components = 0;
    for (String method : methods) {
        if (!visited.contains(method)) {
            dfs(graph, method, visited);
            components++;
        }
```

```
        }
        return components;
    }
    private void dfs(Map<String, Set<String>> graph, String method, Set<String> visited) {
        Stack<String> stack = new Stack<>();
        stack.push(method);
        while (!stack.isEmpty()) {
            String current = stack.pop();
            if (!visited.contains(current)) {
                visited.add(current);
                for (String neighbor : graph.get(current)) {
                    if (!visited.contains(neighbor)) {
                        stack.push(neighbor);
                    }
                }
            }
        }
    }
}
public static void main(String[] args) {
    ClassInfo classInfo = new ClassInfo();
```

```
        // Ejemplo de métodos y atributos
        classInfo.addMethod("method1", new HashSet<>(Arrays.asList("attr1", "attr2")));
        classInfo.addMethod("method2", new HashSet<>(Arrays.asList("attr2")));
        classInfo.addMethod("method3", new HashSet<>(Arrays.asList("attr3")));
        int lcom4 = classInfo.calculateLCOM4();
        System.out.println("LCOM4 = " + lcom4);
    }
}
```

Ejercicio:

1. Analiza el nivel de cohesión de una clase Java proporcionada mediante la métrica LCOM4.
2. Identifica problemas de diseño basados en la métrica calculada.
3. Refactoriza la clase para mejorar su cohesión y reducir el valor de LCOM4.

Utiliza esta clase:

```
public class CineManager {
    private List<Pelicula> peliculas;
    private Map<Integer, Sala> salas;

    public CineManager() {
        peliculas = new ArrayList<>();
        salas = new HashMap<>();
    }

    public void agregarPelicula(Pelicula pelicula) {
        peliculas.add(pelicula);
    }
```

```
public boolean eliminarPelicula(String titulo) {
    return peliculas.removeIf(p -> p.getTitulo().equals(titulo));
}

public void agregarSala(Sala sala) {
    salas.put(sala.getId(), sala);
}

public boolean eliminarSala(int id) {
    return salas.remove(id) != null;
}

public List<Sesion> sesionesPorPelicula(String titulo) {
    return peliculas.stream()
                .filter(p -> p.getTitulo().equals(titulo))
                .flatMap(p -> p.getSesiones().stream())
                .collect(Collectors.toList());
}

public void programarSesion(String titulo, Sesion sesion) {
    peliculas.stream()
            .filter(p -> p.getTitulo().equals(titulo))
            .findFirst()
            .ifPresent(p -> p.agregarSesion(sesion));
}

public int contarPeliculas() {
    return peliculas.size();
}
```

```
    public int contarSalas() {
        return salas.size();
    }
}

class Pelicula {
    private String titulo;
    private List<Sesion> sesiones;

    public Pelicula(String titulo) {
        this.titulo = titulo;
        sesiones = new ArrayList<>();
    }

    public void agregarSesion(Sesion sesion) {
        sesiones.add(sesion);
    }

    public String getTitulo() {
        return titulo;
    }

    public List<Sesion> getSesiones() {
        return new ArrayList<>(sesiones);
    }
}

class Sala {
```

```
    private int id;
    private int capacidad;

    public Sala(int id, int capacidad) {
        this.id = id;
        this.capacidad = capacidad;
    }

    public int getId() {
        return id;
    }
}

class Sesion {
    private String horaInicio;
    private int duracion; // Duración en minutos

    public Sesion(String horaInicio, int duracion) {
        this.horaInicio = horaInicio;
        this.duracion = duracion;
    }
}
```

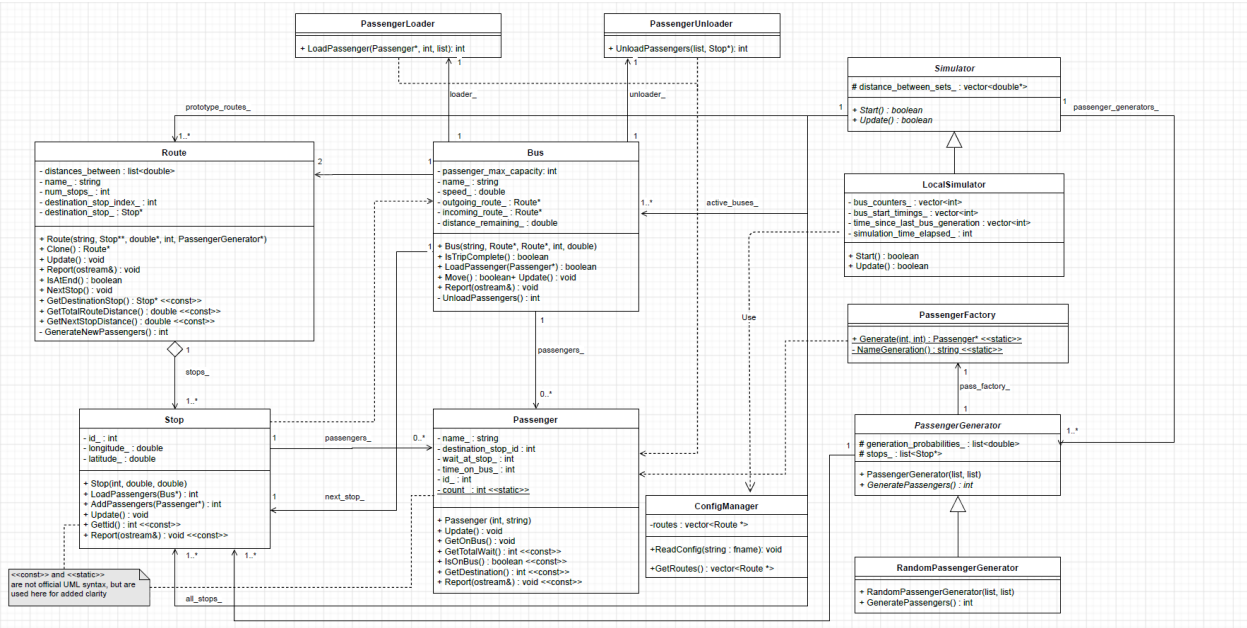
3. Cohesion Among Methods in Class (CAMC)

La métrica CAMC evalúa la cohesión basándose en el tipo de parámetros que los métodos de una clase utilizan. Si todos los métodos de una clase utilizan tipos de datos similares, la clase es considerada altamente cohesiva. Esta métrica proporciona una visión de cuán relacionadas están las operaciones de una clase entre sí.

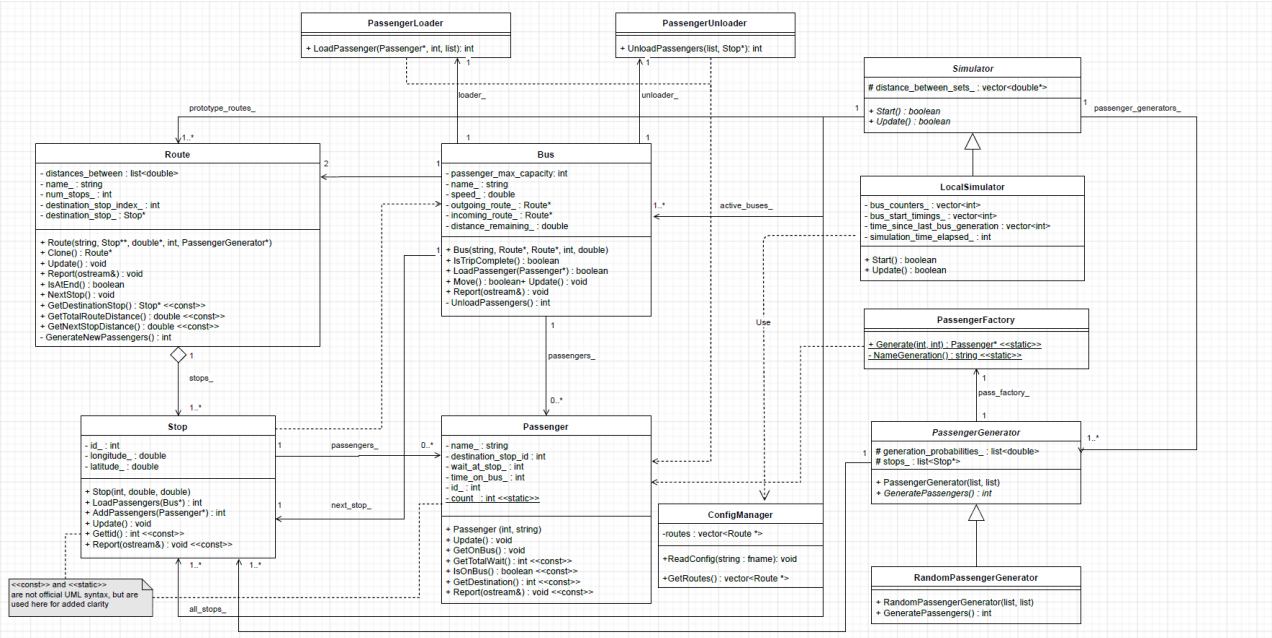
CAMC = [sumatorio de tipos de parámetros únicos utilizados por todos los métodos] /[número total de métodos× número máximo de parámetros por método.]

Ejercicios:

Para el siguiente diagrama de clases, calcula el valor de Inestabilidad para la clase Stop. Usa hasta 3 cifras significativas (por ejemplo, .5 para 1/2 y .333 para 1/3 son aceptables).



Para el siguiente diagrama de clases, calcula el valor de Inestabilidad para la clase PassengerUnloader. Usa hasta 3 cifras significativas (por ejemplo, .5 para 1/2 y .333 para 1/3 son aceptables).



Al calcular LCOM4, ignoramos los constructores y destructores. Los constructores y destructores frecuentemente establecen y eliminan todas las variables en la clase, haciendo que todos los métodos estén conectados a través de estas variables, lo que aumenta la cohesión de manera artificial.

LCOM4 = 1 indica una clase cohesiva, la cual es la "buena" clase.

LCOM4 >= 2 indica un problema. La clase debería dividirse en varias clases más pequeñas.

LCOM4 = 0 ocurre cuando no hay métodos en una clase. Esto también es una "mala" clase.

Calcula el valor de la medición LCOM4 para el siguiente código:

(Nota: Puedes compartir el código que necesitas analizar para que pueda calcular el valor LCOM4).

```
#include "src/route.h"

Route::Route(std::string name, Stop ** stops, double * distances,

    int num_stops, PassengerGenerator * generator) {

  // Los constructores se ignoran en el cálculo de LCOM4

}

void Route::Update() {

  GenerateNewPassengers();

  for (std::list<Stop *>::iterator it = stops_.begin();

      it != stops_.end(); it++) {

    (*it)->Update();

  }

}

bool Route::IsAtEnd() const {

  return destination_stop_index_ >= num_stops_;
```

```
}

void Route::NextStop() {
  destination_stop_index_++;
  if (destination_stop_index_ < num_stops_) {
    std::list<Stop *>::const_iterator iter = stops_.begin();
    std::advance(iter, destination_stop_index_);
    destination_stop_ = *iter;
  } else {
    destination_stop_ = (*stops_.end());
  }
}

Stop * Route::GetDestinationStop() const {
  return destination_stop_;
}

double Route::GetTotalRouteDistance() const {
  int total_distance = 0;
  for (std::list<double>::const_iterator iter = distances_between_.begin();
      iter != distances_between_.end();
      iter++) {
    total_distance += *iter;
  }
```

```
  return total_distance;

}



double Route::GetNextStopDistance() const {

  std::list<double>::const_iterator iter = distances_between_.begin();

  std::advance(iter, destination_stop_index_-1);

  return *iter;  // resolviendo el iterador se obtiene el Stop * de la lista

}

int Route::GenerateNewPassengers() {

  // devolviendo el número de pasajeros añadidos por el generador

  return generator_->GeneratePassengers();

}
```

Para calcular el valor de LCOM4 para el código proporcionado, primero debemos identificar los métodos en la clase **Route** y cómo acceden a las variables compartidas. Recordemos que LCOM4 mide la cohesión basándose en el número de componentes conectados en un grafo donde los nodos son métodos y los arcos existen si los métodos acceden a los mismos atributos.

Aquí está la lista de métodos y su acceso a las variables de instancia en la clase **Route**:

1. **Route (constructor)** - Accede a **stops**, **distances**, **num_stops**, **generator**.
2. **Update()** - Accede a **stops_**, llama a **GenerateNewPassengers()**.
3. **IsAtEnd()** - Accede a **destination_stop_index_**, **num_stops_**.
4. **NextStop()** - Accede a **destination_stop_index_**, **num_stops_**, **stops_**, **destination_stop_**.
5. **GetDestinationStop()** - Accede a **destination_stop_**.
6. **GetTotalRouteDistance()** - Accede a **distances_between_**.
7. **GetNextStopDistance()** - Accede a **distances_between_**, **destination_stop_index_**.
8. **GenerateNewPassengers()** - Accede a **generator_**.

Análisis de la conexión entre métodos:

- **Update()** está conectado con **GenerateNewPassengers()** ya que lo llama directamente.

- **NextStop()**, **IsAtEnd()**, **GetDestinationStop()**, y **GetNextStopDistance()** comparten acceso a **destination_stop_index_**.
- **GetTotalRouteDistance()** y **GetNextStopDistance()** comparten acceso a **distances_between_**.

Cálculo de LCOM4:

- **Update()** y **GenerateNewPassengers()** forman un componente porque están directamente conectados.
- **NextStop()**, **IsAtEnd()**, **GetDestinationStop()**, y **GetNextStopDistance()** forman otro componente debido al acceso compartido a **destination_stop_index_**.
- **GetTotalRouteDistance()** y **GetNextStopDistance()** podrían considerarse parte de un mismo componente o separados; depende de si consideramos que el acceso a **distances_between_** es suficiente para formar un componente.

Dado esto, tenemos al menos dos componentes claramente definidos, lo que sugiere que:

LCOM4 = 2 o más, dependiendo de cómo consideremos las conexiones entre **GetTotalRouteDistance()** y **GetNextStopDistance()**.

LCOM4 >= 2 indica que la clase podría tener problemas de cohesión y podría ser beneficiosa dividirla en clases más pequeñas. Esto es coherente con el análisis, ya que hay distintos grupos de métodos que operan sobre distintas variables y que podrían justificar una separación en clases más especializadas.

Complejidad ciclomática

La complejidad ciclomática mide la complejidad de un programa cuantificando el número de caminos linealmente independientes a través del grafo de flujo de control del programa. En otras palabras, la complejidad ciclomática calcula el número de caminos diferentes que se pueden tomar a través de un código fuente, lo cual proporciona una indicación de cuántas pruebas diferentes son necesarias para cubrir todas las posibles ejecuciones del programa y, por ende, la dificultad de entender y mantener el código.

Funcionamiento de la complejidad ciclomática

La complejidad ciclomática se puede entender mejor considerando cómo se estructura un programa a nivel de flujo de control:

- **Nodos**: Representan las partes del código donde se ejecuta la acción (por ejemplo, ejecución de una instrucción).
- **Aristas**: Representan el flujo de control de una parte del código a otra (por ejemplo, una declaración de salto como un **if** o un **loop**).

Implicaciones de la complejidad ciclomática

- **Mantenimiento y comprensión**: Un valor alto en la complejidad ciclomática indica que el código es potencialmente más difícil de entender y mantener. Es más probable que un módulo con alta complejidad contenga más defectos y sea más difícil de modificar sin introducir errores.
- **Pruebas**: Un valor alto también sugiere la necesidad de más casos de prueba para lograr una cobertura completa del código, lo cual implica que el proceso de prueba puede ser más largo y más costoso.
- **Refactorización**: Los valores altos pueden servir como un indicador para la refactorización del código, sugiriendo que el código podría beneficiarse de ser dividido en funciones o módulos más pequeños y manejables.

Valores típicos

- **0-10**: Generalmente se considera un buen rango de complejidad, indicando que el código es probablemente fácil de entender y mantener.
- **11-20**: Indica una complejidad moderada; aunque aún manejable, el código en este rango podría beneficiarse de cierta refactorización.
- **21+**: Representa una alta complejidad y es una señal de alerta para revisar el diseño del código. A menudo, el código en este rango se beneficia significativamente de ser dividido en componentes más pequeños.

La complejidad ciclomática es ampliamente utilizada en la industria del software para medir la calidad del código y como parte de un análisis estático general durante las revisiones de código y las fases de prueba.

Herramientas de análisis de código como SonarQube, Coverity, y muchos IDEs incorporan calculadoras de complejidad ciclomática para ayudar a los desarrolladores a gestionar la calidad de sus programas de manera más efectiva

La complejidad ciclomática se calcula mediante la fórmula:

$M = E - N + 2P$

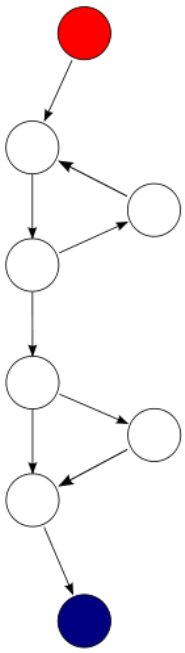
donde:

E = el número de aristas del grafo.

N = el número de nodos del grafo.

P = el número de componentes conectados.

Calcula la complejidad ciclomática para el siguiente grafo de flujo de control:



El grafo representa un diagrama de flujo para la complejidad ciclomática con un total de cinco nodos (los círculos blancos) y dos nodos de entrada/salida (el círculo rojo y azul, respectivamente). La complejidad ciclomática, medida por el número de caminos linealmente independientes a través del código (que en este caso serían los posibles caminos desde el nodo rojo al nodo azul), parece ser relativamente simple.

```
public class FlowExample {
    public static void main(String[] args) {
        int input = Integer.parseInt(args[0]);
        if (input > 0) {
            System.out.println("Input is positive");
            if (input > 100) {
                System.out.println("Input is greater than 100");
            } else {
                System.out.println("Input is not greater than 100");
            }
```

```
        } else {
            System.out.println("Input is not positive");
        }
        System.out.println("Execution complete");
    }
}
```

La complejidad ciclomática se calcula mediante la fórmula:

$M=E-N+2P$

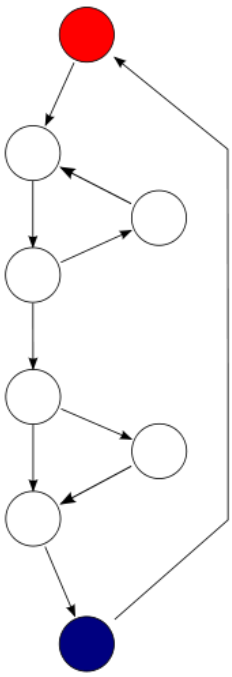
donde:

E = el número de aristas del grafo.

N = el número de nodos del grafo.

P = el número de componentes conectados.

Calcula la complejidad ciclomática para el siguiente grafo de flujo de control:



Un ejemplo de código en Java que podría representar este grafo es:

```
public class ComplexDecisionFlow {
    public static void main(String[] args) {
        int x = Integer.parseInt(args[0]); // Nodo de entrada
        if (x > 10) {  // Primer nodo blanco desde la entrada
            System.out.println("X is greater than 10");
            if (x > 20) {  // Segundo nodo blanco a la derecha
                System.out.println("X is also greater than 20");
            } else {
                System.out.println("X is not greater than 20");
            }
            System.out.println("After checking x > 20");  // Nodo blanco que une las rutas
        } else {
            System.out.println("X is not greater than 10");  // Segundo nodo blanco a la izquierda
        }
        System.out.println("End of the flow");  // Nodo de salida
    }
}
```

La complejidad ciclomática se calcula mediante la fórmula:

$M = E - N + 2P$

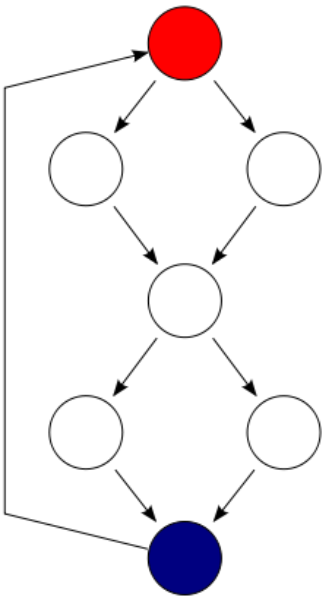
donde:

E = el número de aristas del grafo.

N = el número de nodos del grafo.

P = el número de componentes conectados.

Calcula la complejidad ciclomática para el siguiente grafo de flujo de control:



Este grafo representa un flujo de control con tres decisiones binarias y puntos donde las rutas convergen antes de llegar al nodo final. A continuación. Un ejemplo de código en Java que podría representar este grafo es:

```
public class DecisionTree {

    public static void main(String[] args) {

        int a = Integer.parseInt(args[0]); // Nodo de entrada (rojo)
```

```
        if (a > 10) {  // Primer nodo blanco
            if (a > 20) {  // Segundo nodo blanco, parte superior
                System.out.println("A is greater than 20");
            } else {
                System.out.println("A is greater than 10 but less than or equal to 20");
            }
            // Punto donde convergen las rutas después de la primera decisión
            System.out.println("A is greater than 10");
        } else {
            System.out.println("A is 10 or less");  // Segundo nodo blanco, parte inferior
        }
        // Nodo de salida (azul)
        System.out.println("End of program");
    }
}
```

Ejercicio 1: Análisis de complejidad ciclomática

Dado el siguiente fragmento de código Java:

```
public class Analysis {
    public static void process(int number) {
        if (number > 0) {
            if (number % 2 == 0) {
                System.out.println("Positive even number");
```

```
            } else {
                System.out.println("Positive odd number");
            }
        } else if (number < 0) {
            if (number % 2 == 0) {
                System.out.println("Negative even number");
            } else {
                System.out.println("Negative odd number");
            }
        } else {
            System.out.println("Number is zero");
        }
    }
}
```

- Calcula la complejidad ciclomática del método **process**.
- Dibuja el grafo de flujo de control para este método.
- Identifica todos los caminos independientes en el código.

Ejercicio 2: Refactorización para reducir complejidad

Refactoriza el siguiente código para reducir su complejidad ciclomática, manteniendo la misma funcionalidad:

```
public class Refactor {
    public static void main(String[] args) {
        int a = Integer.parseInt(args[0]);
```

```
    if (a > 50) {
      if (a > 100) {
        System.out.println("A is greater than 100");
      } else {
        System.out.println("A is greater than 50 but not more than 100");
      }
    } else {
      if (a < 20) {
        System.out.println("A is less than 20");
      } else {
        System.out.println("A is between 20 and 50");
      }
    }
  }
}
```

Ejercicio 3: Diseño de grafo de flujo de control

Escribe un programa Java que se ajuste al siguiente grafo de flujo de control:

- Un nodo de inicio.
- Dos decisiones consecutivas que bifurcan el flujo en cuatro caminos.
- Cuatro nodos finales, uno para cada posible camino.

Proporciona el código Java y dibuja el grafo correspondiente.

Ejercicio 4: Test de camino de vases

Para el código Java que desarrollaste en el ejercicio 3: enumera todos los casos de prueba necesarios para cubrir todos los caminos del grafo de flujo de control basados en la prueba de caminos de bases.

Problemas

Desarrolla una herramienta en Java que analice un conjunto de clases Java y calcule la complejidad ciclomática de cada método.

- Parsear el código fuente Java para identificar métodos y sus flujos de control (if, for, while, switch, etc.).
- Calcular la complejidad ciclomática según la fórmula mencionada previamente.
- Genera un reporte que indique la complejidad ciclomática de cada método y sugiera puntos de refactorización para métodos con alta complejidad.

Crea una herramienta que mida el acoplamiento eferente y aferente de las clases en un proyecto Java.

- Analizar las dependencias entre las clases de un proyecto Java utilizando análisis estático del código.
- Calcular el acoplamiento eferente y aferente para cada clase.
- Proveer recomendaciones para reducir el acoplamiento donde sea necesario.

Implementar un sistema que evalúe la cohesión de las clases en un proyecto Java utilizando LCOM4.

- Identifica y clasifica los métodos y variables de instancia de cada clase.
- Calcula LCOM4 para cada clase.
- Sugiere modificaciones o refactorizaciones para mejorar la cohesión de clases con bajo LCOM4.

Desarrolla una herramienta integrada que proporcione análisis de múltiples métricas de software para proyectos Java.

- Combina las funcionalidades de los ejercicios anteriores en una sola herramienta.
- Añade funcionalidades de interfaz de usuario para permitir a los usuarios seleccionar proyectos y ver los resultados de manera gráfica.
- Implementar exportación de los resultados a formatos como CSV o PDF para análisis posterior.

Simular los efectos de diversas refactorizaciones en las métricas de un proyecto Java.

- Permitir al usuario seleccionar refactorizaciones específicas (como dividir clases, extraer métodos, etc.).
- Mostrar cómo esas refactorizaciones afectarían las métricas de complejidad, acoplamiento y cohesión.
- Evaluar el impacto de las refactorizaciones propuestas antes de aplicarlas en el código real.

Sugerencia:

Pasos para desarrollar el simulador de refactorización

Paso 1: Análisis de código base

Primero, necesitas una forma de analizar el código base para identificar oportunidades de refactorización. Esto podría incluir identificar métodos largos, clases con alta complejidad ciclomática, bajo acoplamiento o baja cohesión, etc.

Paso 2: Sugerencia de refactorizaciones

Basado en el análisis, tu herramienta debe ser capaz de sugerir diferentes tipos de refactorizaciones. Por ejemplo:

- Extraer método (para reducir la complejidad ciclomática).
- Mover método (para mejorar la cohesión o reducir el acoplamiento).
- Dividir clase (para clases con demasiadas responsabilidades).

Paso 3: Simulación de refactorización

Antes de aplicar los cambios, simula el efecto que tendrán estas refactorizaciones en las métricas del código. Esto podría hacerse mediante un modelo de predicción o estimaciones basadas en heurísticas conocidas.

Paso 4: Evaluación y reporte

Evalúa los posibles cambios en las métricas después de la refactorización simulada y presenta los resultados al usuario, permitiendo comparar el "antes" y el "después".

Ejemplo de código base

Aquí tienes un ejemplo muy básico de cómo podrías estructurar parte del código para este ejercicio, utilizando pseudo-código y Java:

```
import com.github.javaparser.JavaParser;
import com.github.javaparser.ast.CompilationUnit;
import com.github.javaparser.ast.body.MethodDeclaration;
import com.github.javaparser.ast.visitor.VoidVisitorAdapter;

import java.io.FileInputStream;
import java.util.ArrayList;
import java.util.List;

public class RefactoringSimulator {

    public static void main(String[] args) throws Exception {
        FileInputStream in = new FileInputStream("Path/To/Your/JavaFile.java");
        CompilationUnit cu = JavaParser.parse(in);

        List<RefactoringSuggestion> suggestions = analyzeCode(cu);
        List<RefactoringImpact> impacts = simulateRefactorings(suggestions, cu);

        reportResults(impacts);
    }

    static List<RefactoringSuggestion> analyzeCode(CompilationUnit cu) {
        // Implement analysis logic here
        return new ArrayList<>();
    }

    static List<RefactoringImpact> simulateRefactorings(List<RefactoringSuggestion> suggestions,
CompilationUnit cu) {
```

```
        // Simulate refactorings and calculate potential impact on metrics
        return new ArrayList<>();
    }

    static void reportResults(List<RefactoringImpact> impacts) {
        // Output the simulation results
        impacts.forEach(impact -> System.out.println(impact));
    }
}

class RefactoringSuggestion {
    // Details about suggested refactorings
}

class RefactoringImpact {
    // Details about the impact of a refactoring
}
```