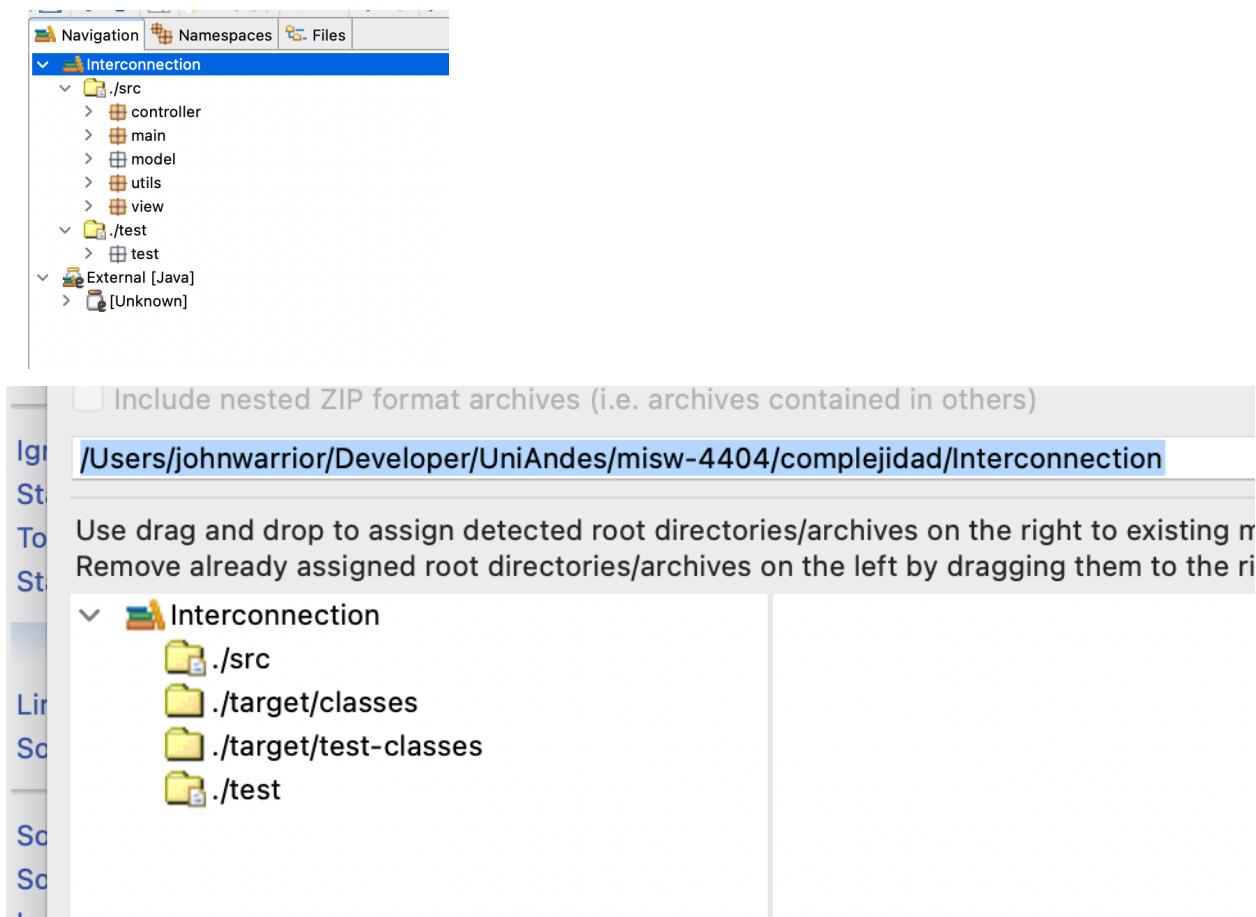
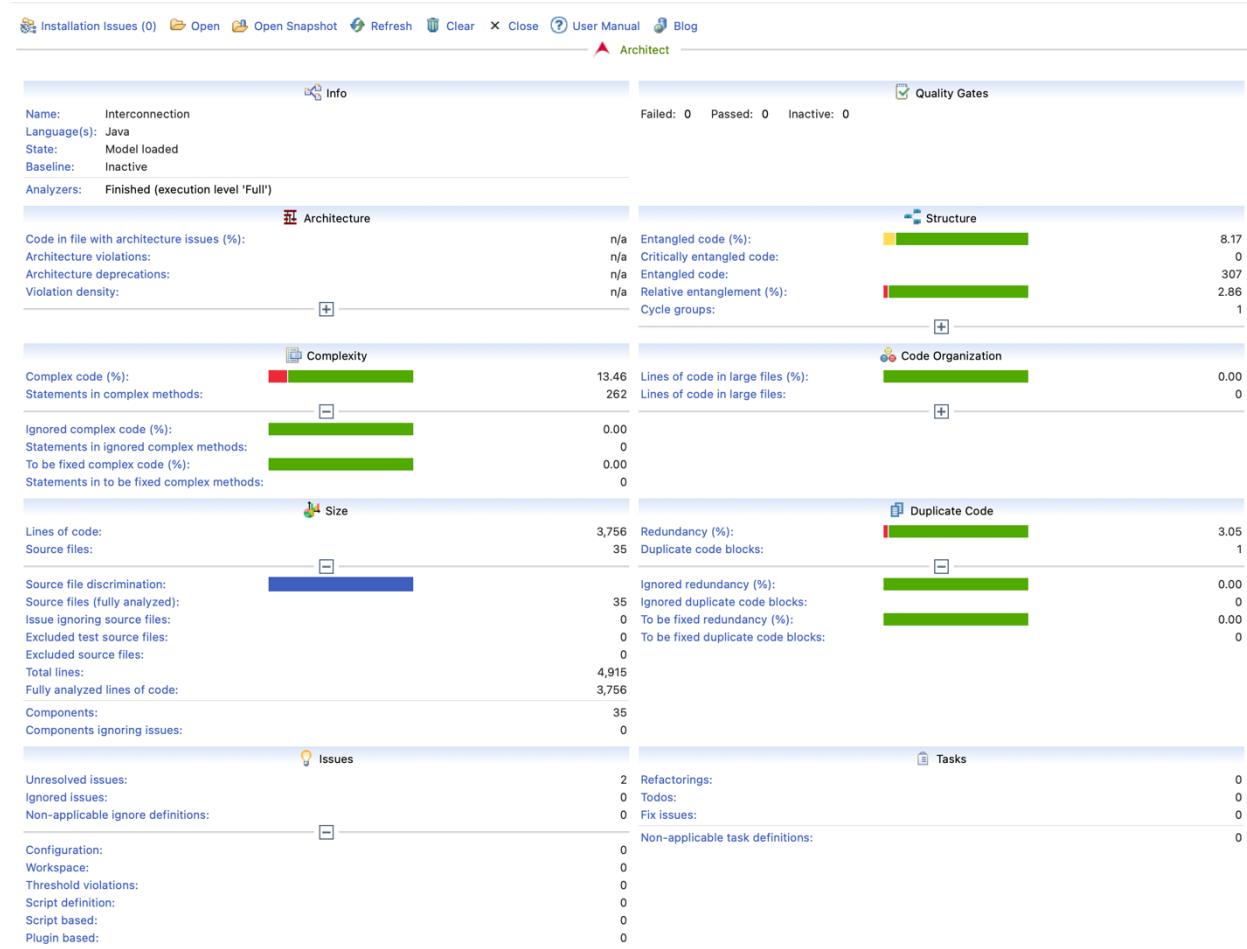


# Proyecto análisis de calidad

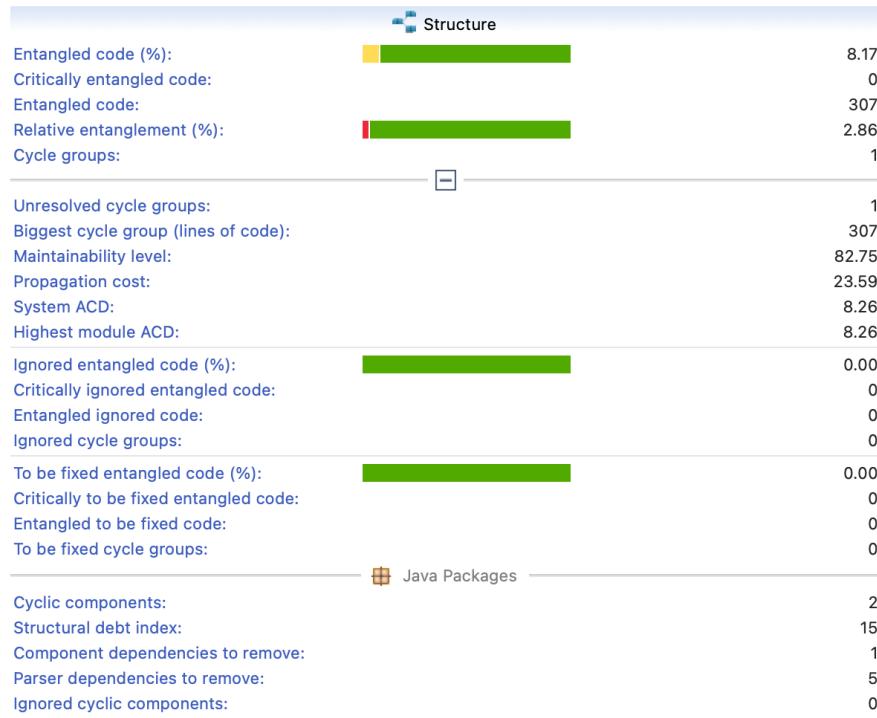
## Configuración de proyecto en Sonar Graph



# Análisis estructural del proyecto



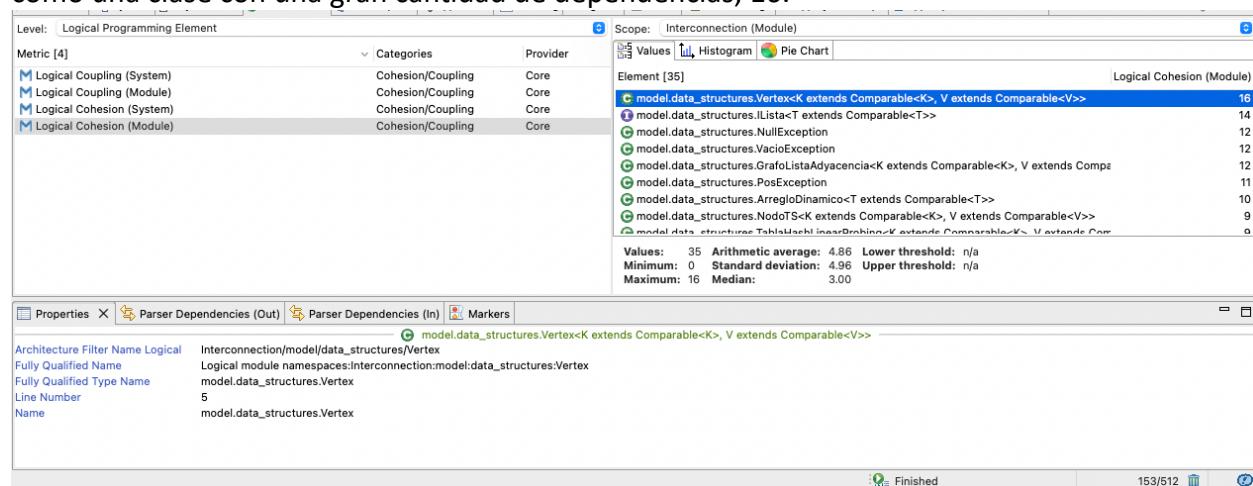
## Análisis de acoplamiento



Para este programa, el Average component dependency (ACD) tiene un valor de 8.26, lo que significa que, en promedio, cualquier modificación a un archivo se estarán propagando en otros 8.26 archivos.

Teniendo en cuenta el programa se encuentra compuesto por 35 archivos y 4.915 líneas de código totales en el programa. Y tomando la referencia un único grupo cíclico incluyendo 307 líneas de código, deducimos que se produce un impacto del 6.24% de la base de código cada vez que se realiza un cambio en el programa.

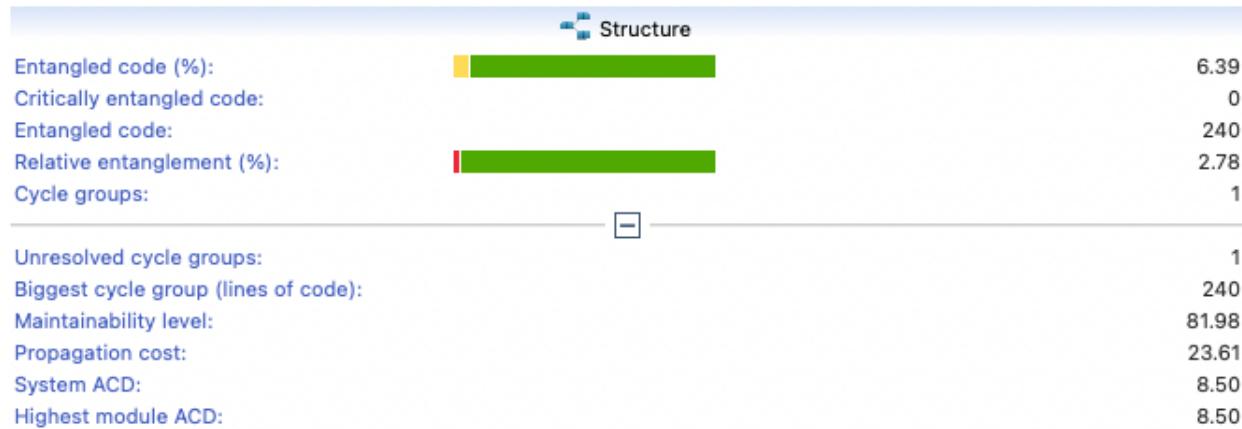
Revisando las métricas dentro de “Logical Programming Element” podemos ver la clase “Vertex” como una clase con una gran cantidad de dependencias, 16.



## Refactoring

### Acoplamiento y cohesión

Dado que el resultado del primer análisis fue una marcada dependencia hacia la clase Vertex, se realizó la separación de responsabilidades extrayendo de ella la parte encargada de realizar la navegación entre vértices.



No se obtuvo una mejora significativa en el ACD, pero se disminuyó la métrica de código enredado, así. Como una mejora en la cohesión de esta clase con respecto a los otros elementos.

Se creó la clase GraphAlgorithms, para tomar estas responsabilidades:

```
package model.data_structures;

2 usages new *
public class GraphAlgorithms<K extends Comparable<K>, V extends Comparable<V>> {
    1 usage new *
    >     public void bfs(Vertex<K, V> startVertex) [...]

    2 usages new *
    >     public void dfs(Vertex<K, V> currentVertex) [...]

    1 usage new *
    >     public void topologicalOrder(ILista<Vertex<K, V>> vertices, ColaEncadenada<Vertex<K, V>> |
        2 usages new *
    >         private void dfsTopologicalOrder(Vertex<K, V> vertex, ColaEncadenada<Vertex<K, V>> pre, Co
    }
```

Continuando con el refactoring, se logra extraer la mayor cantidad de funcionalidad ajena a **Vertex**, obteniendo la siguiente estructura para esta clase:

The screenshot shows a Java code editor with the following code content:

```
package model.data_structures;

import java.util.Comparator;

100 usages ▲ ncardozo *
public class Vertex<K extends Comparable<K>, V extends Comparable <V>> implements Comparable<Vertex<K, V>>
{
    3 usages
    private K key;
    2 usages
    private V value;
    3 usages
    private ILista<Edge<K, V>> arcos;
    3 usages
    private boolean marked;

    1 usage ▲ ncardozo
    public Vertex(K id, V value)
    {
        ...
    }

    39 usages ▲ ncardozo
    public K getId() { return key; }

    21 usages ▲ ncardozo
    public V getInfo() { return value; }

    10 usages ▲ ncardozo
    public boolean getMark() { return marked; }

    6 usages ▲ ncardozo
    public void mark() { marked=true; }

    2 usages ▲ ncardozo
    public void unmark() { marked=false; }

    no usages ▲ ncardozo
    public int getOutdegree()
    {
        return arcos.size();
    }

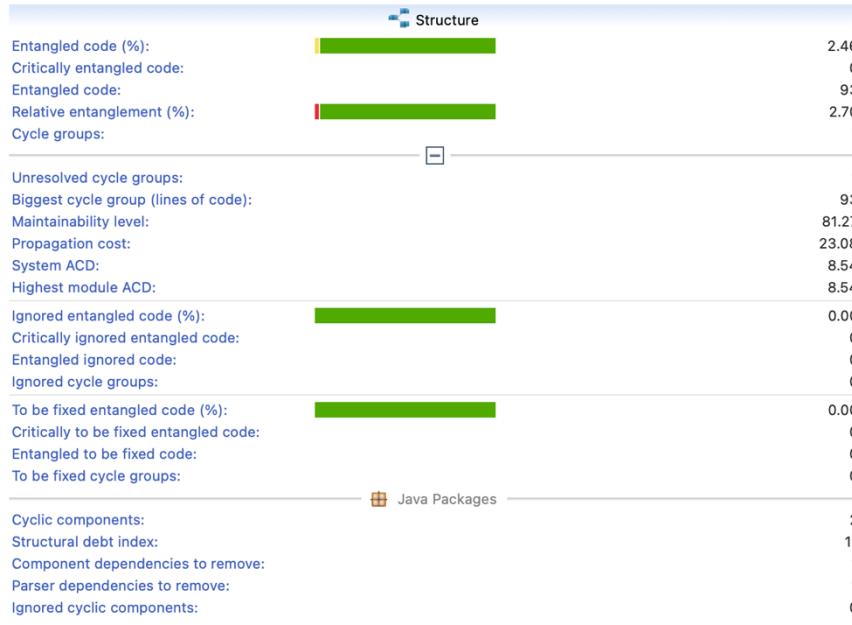
    no usages ▲ ncardozo
    public int indegree() { return arcos.size(); }

    ▲ ncardozo
    @Override
    public int compareTo(Vertex<K, V> o) { return key.compareTo(o.getId()); }

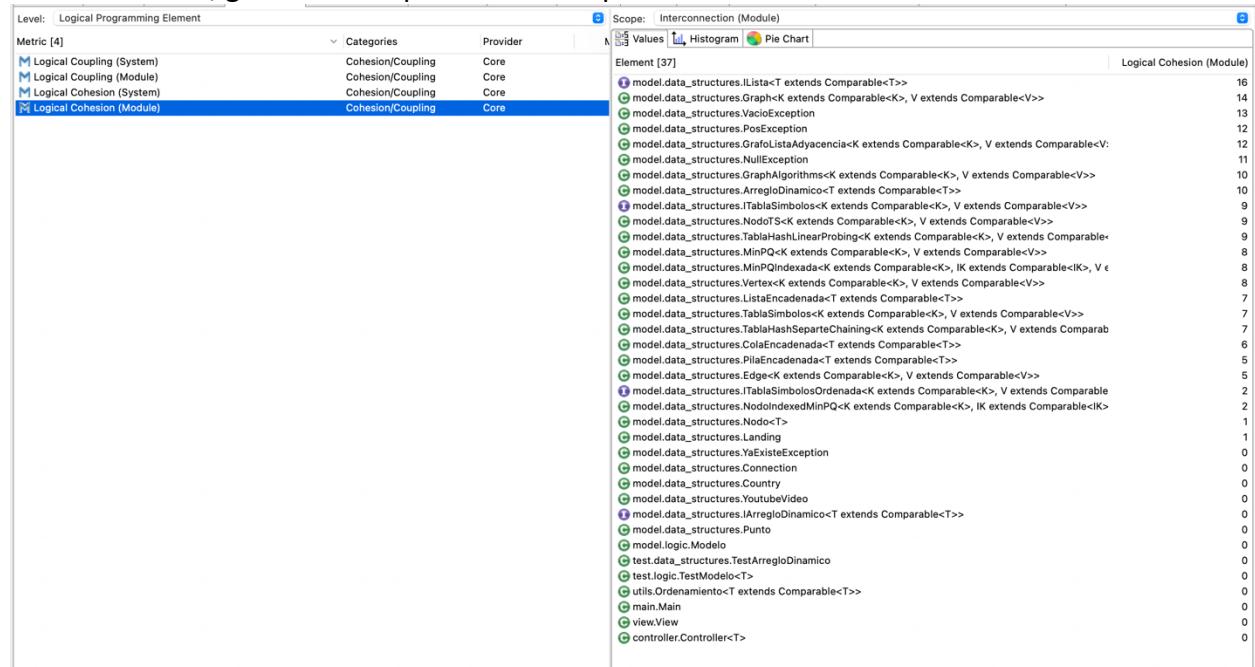
    2 usages ▲ ncardozo
    public static class ComparadorXKey implements Comparator<Vertex<String, Landing>>
    {
        ...
    }
}
```

Permitiendo la siguiente mejora en cuanto a la cantidad de código acoplado y cohesión:

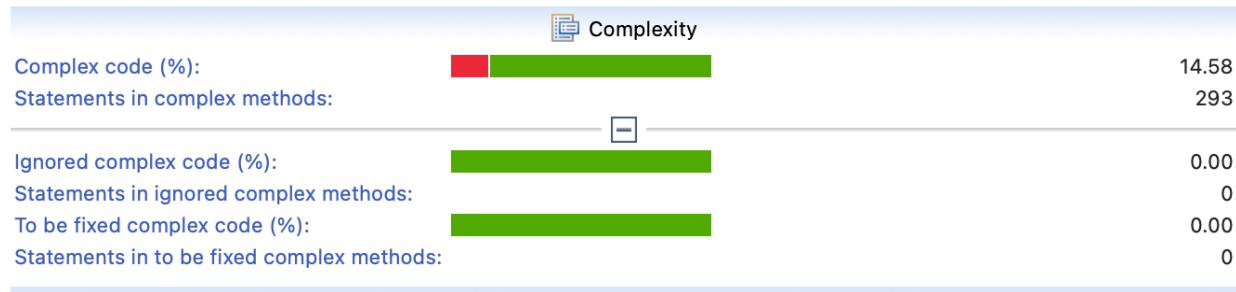
En cuanto al porcentaje de código enredado, se disminuye considerablemente el porcentaje presentado en el proyecto.



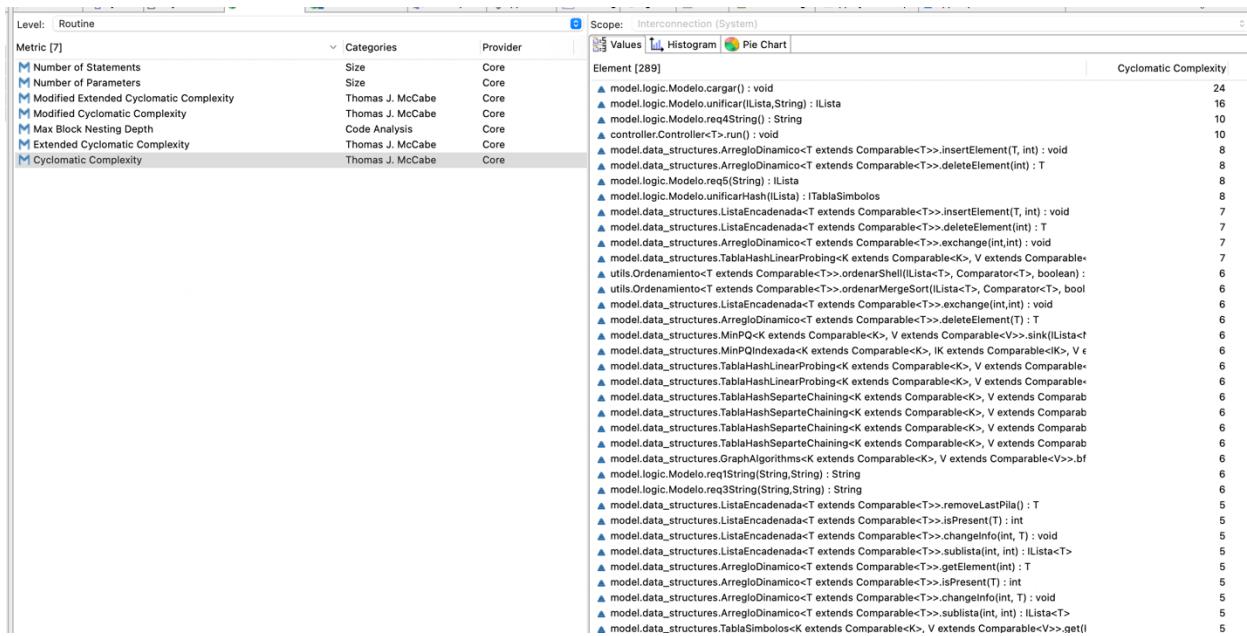
En cuanto a la métrica de cohesión disminuyo la cantidad de dependencias que presentaba la clase a la mitad, gracias a la separación de responsabilidades.



## Complejidad



En orden de cumplir con el perfil de calidad relacionado con la complejidad del código del proyecto, realizamos un análisis de la complejidad ciclomática, dado que en el momento no cumplimos con las métricas solicitadas. Dicho análisis arroja los siguientes puntos como a tener en cuenta para resolver la complejidad ciclomática del proyecto:



A simple vista se puede identificar que la mayoría del código con alta complejidad se encuentra en la clase Modelo.

- Procedemos a disminuir la complejidad de la función cargar, separando la lógica que realiza la carga de los diferentes archivos CSV de insumo para procesar los grafos.

The screenshot shows a Java code editor with a dark theme. The code in the editor is as follows:

```

1 usage  ncardozo *
public void cargar() throws IOException
{
    grafo= new GrafoListaAdyacencia( numVertices: 2);
    paises= new TablaHashLinearProbing( tamInicial: 2);
    points= new TablaHashLinearProbing( tamInicial: 2);
    landingidtabla= new TablaHashSeparateChaining( tamInicial: 2);
    nombrecodigo=new TablaHashSeparateChaining( tamInicial: 2);

    loadCountriesData();
    loadLandingPointsData();
    loadConnectionsData();

    try
    {
        ILista valores = landingidtabla.valueSet();

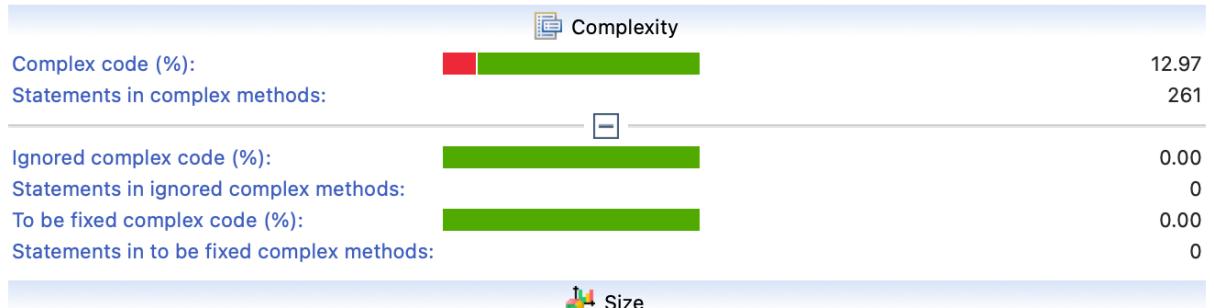
        for(int i=1; i<=valores.size(); i++)
        {
            for(int j=1; j<=((ILista) valores.getElement(i)).size(); j++)
            {
                Vertex vertice1;
                if((ILista) valores.getElement(i) != null)
                {
                    vertice1= (Vertex) ((ILista) valores.getElement(i)).getElement(j);
                }
            }
        }
    }
}

```

Below the code editor is a table showing the cyclomatic complexity of various methods. The table has two columns: 'Method [292]' and 'Cyclomatic Complexity'.

Method [292]	Cyclomatic Complexity
model.logic.Modelo.unificar(ILista, String) : ILista	16
model.logic.Modelo.loadConnectionsData() : void	16
model.logic.Modelo.req4String() : String	10
controller.Controller<T>.run() : void	10
model.data_structures.ArregloDinamico<T extends Comparable<T>>.insertElement(T, int) : void	8
model.data_structures.ArregloDinamico<T extends Comparable<T>>.deleteElement(int) : T	8
model.logic.Modelo.req5(String) : ILista	8
model.logic.Modelo.unificarHash(ILista) : ITablaSimbolos	8
model.data_structures.ListaEncadenada<T extends Comparable<T>>.insertElement(T, int) : void	7
model.data_structures.ListaEncadenada<T extends Comparable<T>>.deleteElement(int) : T	7
model.data_structures.ArregloDinamico<T extends Comparable<T>>.exchange(int,int) : void	7
model.data_structures.TablaHashLinearProbing<K extends Comparable<K>, V extends Comparable<V>>.ordenamiento<T extends Comparable<T>>.ordenarShell(ILista<T>, Comparator<T>, boolean) : void	6
utils.Ordenamiento<T extends Comparable<T>>.ordenarMergeSort(ILista<T>, Comparator<T>, boolean)	6
model.data_structures.ListaEncadenada<T extends Comparable<T>>.exchange(int,int) : void	6
model.data_structures.ArregloDinamico<T extends Comparable<T>>.deleteElement(T) : T	6
model.data_structures.MinPQ<K extends Comparable<K>, V extends Comparable<V>>.sink(ILista<Node>)	6
model.data_structures.MinPQ<Indexada<K extends Comparable<K>, IK extends Comparable<IK>, V extends Comparable<V>>.removeLastPila()	6
model.data_structures.TablaHashLinearProbing<K extends Comparable<K>, V extends Comparable<V>>.sublista(int, int) : ILista<T>	6
model.data_structures.TablaHashSeparateChaining<K extends Comparable<K>, V extends Comparable<V>>.getElement(int) : T	6
model.data_structures.TablaHashSeparateChaining<K extends Comparable<K>, V extends Comparable<V>>.isPresent(T) : int	6
model.logic.Modelo.cargar() : void	6
model.data_structures.ListaEncadenada<T extends Comparable<T>>.removeLastPila() : T	5
model.data_structures.ListaEncadenada<T extends Comparable<T>>.isPresent(T) : int	5
model.data_structures.ListaEncadenada<T extends Comparable<T>>.changeInfo(int, T) : void	5
model.data_structures.ArregloDinamico<T extends Comparable<T>>.getElement(int) : T	5
model.data_structures.ArregloDinamico<T extends Comparable<T>>.isPresent(T) : int	5
model.data_structures.ArregloDinamico<T extends Comparable<T>>.changeInfo(int, T) : void	5

Esta modificación permite disminuir la complejidad tanto del método como del proyecto en general.



2. Se realiza la simplificación de los condicionales en la función unificar, reduciéndose a:

```
2 usages  ▲ ncardozo *
public ILista unificar(ILista lista, String criterio) {
    ILista lista2 = new ArregloDinamico( max: 1);

    Comparator comparador = null;
    Ordenamiento ordenamiento = new Ordenamiento();

    if (criterio.equals("Vertice")) {
        comparador = new Vertex.ComparadorXKey();
    } else {
        comparador = new Country.ComparadorXNombre();
    }

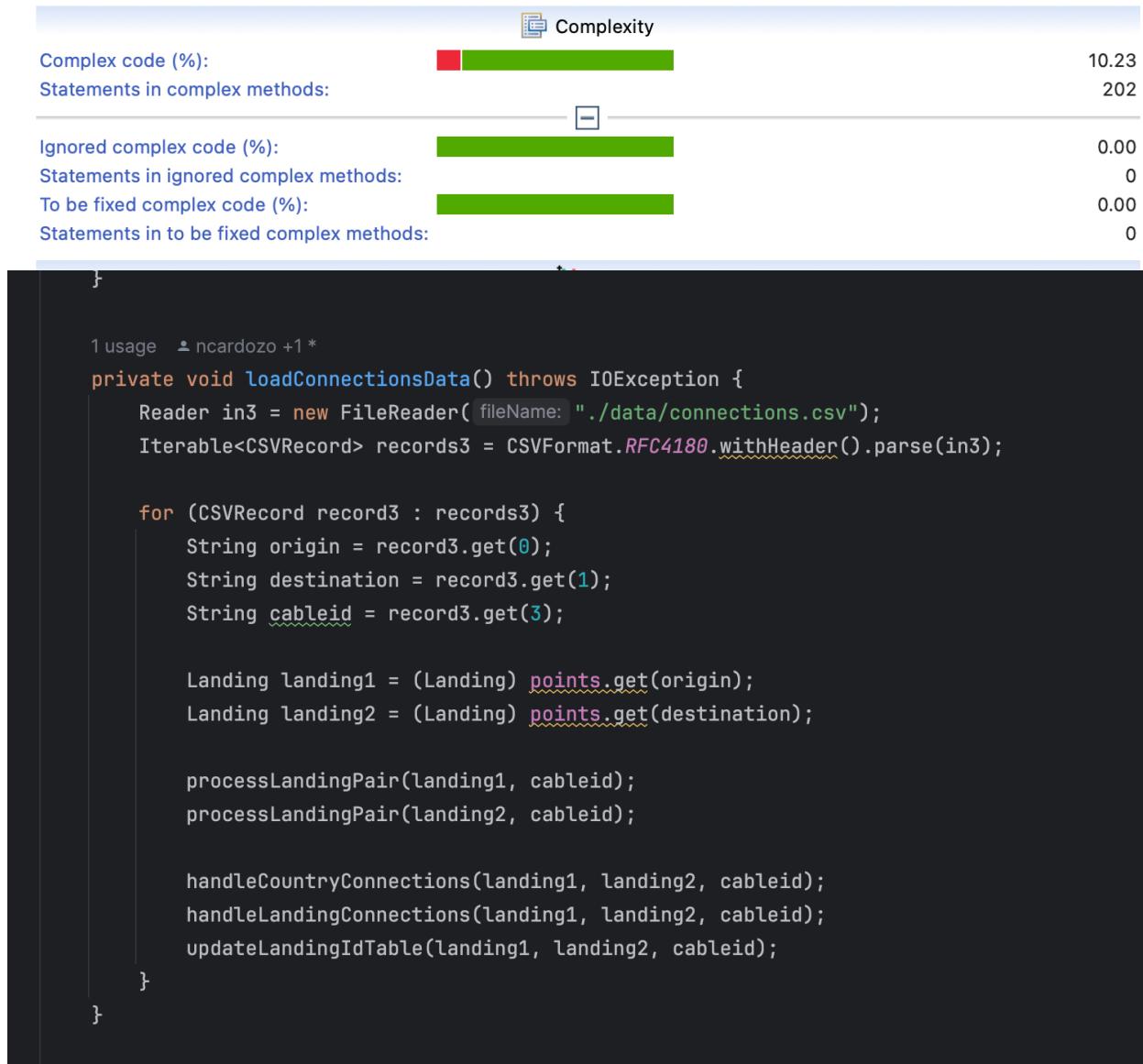
    try {
        if (lista != null && !lista.isEmpty()) {
            ordenamiento.ordenarMergeSort(lista, comparador, ascendente: false);

            for (int i = 1; i <= lista.size(); i++) {
                Object actual = lista.getElement(i);
                Object siguiente = (i < lista.size()) ? lista.getElement( pos: i + 1 ) : null;

                if (siguiente == null || comparador.compare(actual, siguiente) != 0) {
                    lista2.insertElement((Comparable) actual, pos: lista2.size() + 1);
                }
            }
        }
    } catch (PosException | VacioException | NullException e) {
        e.printStackTrace();
    }

    return lista2;
}
```

Este cambio también permitió disminuir la complejidad del proyecto



model.data_structures.TablaHashLinearProbing<K extends Comparable<K>, V extends Comparable<V>	6
model.data_structures.TablaHashLinearProbing<K extends Comparable<K>, V extends Comparable<V>	6
model.data_structures.TablaHashSeparateChaining<K extends Comparable<K>, V extends Comparable<V>	6
model.data_structures.TablaHashSeparateChaining<K extends Comparable<K>, V extends Comparable<V>	6
model.data_structures.TablaHashSeparateChaining<K extends Comparable<K>, V extends Comparable<V>	6
model.data_structures.TablaHashSeparateChaining<K extends Comparable<K>, V extends Comparable<V>	6
model.data_structures.TablaHashSeparateChaining<K extends Comparable<K>, V extends Comparable<V>	6
model.data_structures.GraphAlgorithms<K extends Comparable<K>, V extends Comparable<V>>.bfs(K	6
model.logic.Modelo.req1String(String, String) : String	6
model.logic.Modelo.req3String(String, String) : String	6
<b>model.logic.Modelo.unificar(ILista, String) : ILista</b>	<b>6</b>
model.logic.Modelo.cargar() : void	6
model.data_structures.ListaEncadenada<T extends Comparable<T>>.removeLastPila() : T	5
model.data_structures.ListaEncadenada<T extends Comparable<T>>.isPresent(T) : int	5
model.data_structures.ListaEncadenada<T extends Comparable<T>>.changeInfo(int, T) : void	5
model.data_structures.ListaEncadenada<T extends Comparable<T>>.sublista(int, int) : ILista<T>	5
model.data_structures.ArregloDinamico<T extends Comparable<T>>.getElement(int) : T	5
model.data_structures.ArregloDinamico<T extends Comparable<T>>.isPresent(T) : int	5

3. Se realiza la extracción de funcionalidad de la función loadConnectionsData, reduciéndose a la siguiente función:

```

1 usage  ↵ ncardozo +1 *
private void loadConnectionsData() throws IOException {
    Reader in3 = new FileReader( fileName: "./data/connections.csv");
    Iterable<CSVRecord> records3 = CSVFormat.RFC4180.withHeader().parse(in3);

    for (CSVRecord record3 : records3) {
        String origin = record3.get(0);
        String destination = record3.get(1);
        String cableid = record3.get(3);

        Landing landing1 = (Landing) points.get(origin);
        Landing landing2 = (Landing) points.get(destination);

        processLandingPair(landing1, cableid);
        processLandingPair(landing2, cableid);

        handleCountryConnections(landing1, landing2, cableid);
        handleLandingConnections(landing1, landing2, cableid);
        updateLandingIdTable(landing1, landing2, cableid);
    }
}

```

A su vez disminuyendo la complejidad ciclomática del proyecto significativamente, así como la complejidad de Modelo:

