



UNIVERSIDAD
POLITÉCNICA
DE MADRID

Práctica de análisis dinámico

Integrantes del equipo:

- Diego Machuca Municio
- Diego Allain Martini
- Junyong Li
- Jose Carlos Gago Hernández
- David Ramajo Fernández

Fecha de entrega:

- 13 de diciembre de 2020

Asignatura:

- Verificación y Validación de Software

Enlace al repositorio de Gitlab del grupo

<https://gitlab.com/diegoallainmartini/vv-iwt41-4>

Codacy e informe de cobertura inicial

Al realizar el análisis estático del proyecto base por primera vez, Codacy indica que hay un total de 6 errores como se muestra a continuación.

Issues breakdown



La mayoría de los errores corresponden a **imports** no utilizados y a la utilización de mayúsculas en el nombre de los paquetes. Se han arreglado estos errores durante la realización del proyecto.

src/main/java/singleList/editText/Editor.java

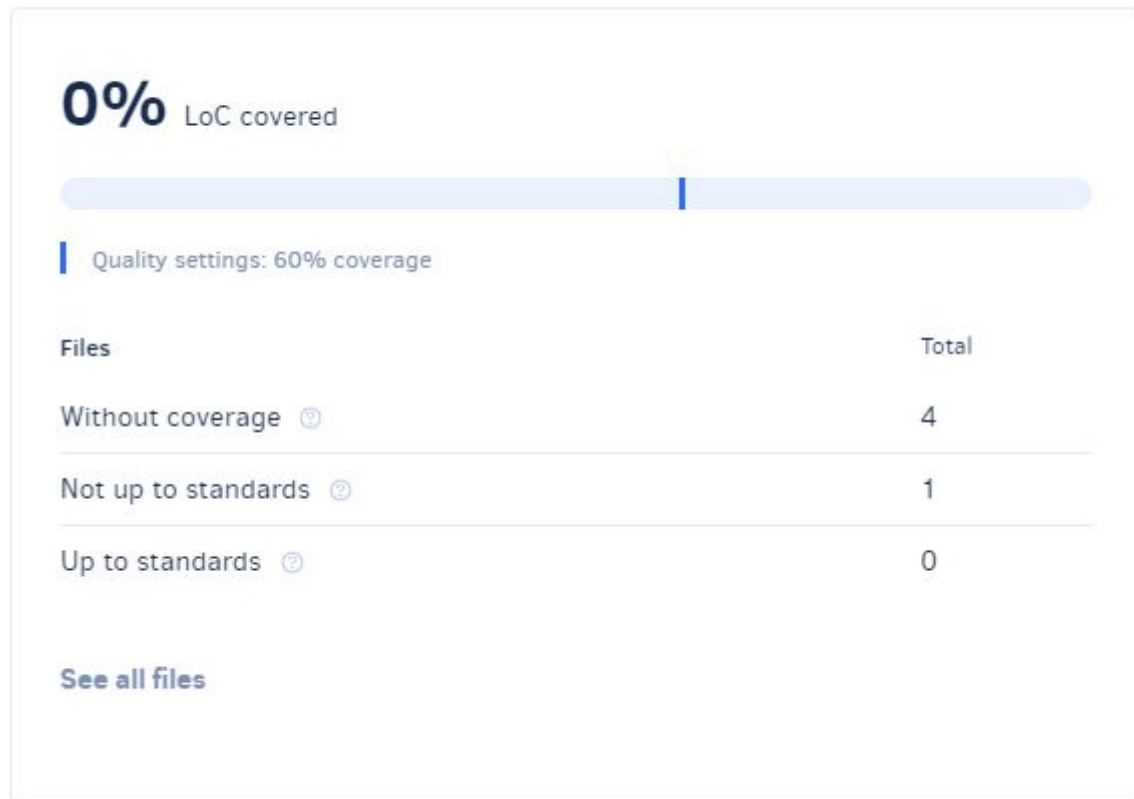
Package name contains upper case characters	▼
1 package singleList.editText;	
Avoid unused imports such as 'singleList.SingleLinkedList'	▼
9 import singleList.SingleLinkedList;	
Unused import - singleList.SingleLinkedList.	▼
9 import singleList.SingleLinkedList;	
Avoid reassigning parameters such as 'inicio'	▼
93 public int numPalabras(int inicio, int fin, String palabra) {	

src/test/java/singleList/editText/SingleLinkedListTest.java

Package name contains upper case characters	▼
1 package singleList.editText;	
Avoid unused imports such as 'org.junit.Assert'	▼
3 import static org.junit.Assert.*;	

Finalmente hay que configurar el informe de cobertura, que se genera configurando el fichero **.yaml**. En la siguiente imagen se muestra el primer informe de cobertura que se ha generado (sin haber solucionado aún los 6 problemas mencionados anteriormente)

Coverage



Tras solucionar los errores que Codacy indicaba, el informe pudo realizarse correctamente. A continuación se muestra la imagen del informe de cobertura realizado mediante Eclipse:

editText (22-nov-2020 13:09:07)				
Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
> editText	<div><div></div></div> 5,6 %	19	322	341

Tests de caja negra

En los test de caja negra hemos decidido usar la metodología *Worst Case for Robustness*, que permite probar todas las posibles de clases de equivalencia, para asegurar el cubrimiento de todos los posibles tipos de entrada a los métodos.

- `public void addAtPos(T element, int p)`

Clases de equivalencia:

Entrada	Regla	Clase válida	Clase no válida
Elemento	números naturales entre 1 y 5000	1. Rango de $1 \leq \text{elemento} \leq 5000$ 2. Elemento = 1 3. Elemento = 2 4. Elemento = 4999 5. Elemento = 5000	6. Elemento < 1 7. Elemento > 5000 8. Elemento = 5001 9. Elemento = 0 10. Elemento no es número natural
Entero	$p \geq 1$	11. $p \geq 10$ 12. $p = 1$ 13. $p = 2$	14. $p < 1$ 15. $p = 0$ 16. p no es entero

Casos de prueba:

Prueba	Entrada	Clases cubiertas	Salida esperada	Salida
1	element = 200 $p = 10$	1, 11	Insertado	Insertado
2	element = 1 $p = 1$	2, 12	Insertado	Insertado
3	element = 2 $p = 2$	3, 13	Insertado	Insertado
4	element = 4999 $p = 10$	4, 11	Insertado	Insertado
5	element = 5000 $p = 10$	5, 11	Insertado	Insertado
6	element = -200 $p = 10$	6, 11	No insertado	Insertado
7	element = 5200 $p = 10$	7, 11	No insertado	Insertado

8	element = 5001 p = 10	8, 11	No insertado	Insertado
9	element = 0 p = 10	9, 11	No insertado	Insertado
10	element = "abc" p = 10	10, 11	No aceptado	No aceptado
11	element = 200 p = -10	1, 14	IllegalArgument Exception	Insertado
12	element = 200 p = 0	1, 15	IllegalArgument Exception	Insertado
13	element = 200 p = "abc"	1, 16	No aceptado	No aceptado

El método addAtPos agrega los elementos en el lugar en el que se espera, sin embargo, al pasarle una posición menor o igual a 0, no se lanza la excepción que se esperaba, sino que es agregado a la lista en la última posición. Finalmente, aquellos elementos que no deberían de añadirse porque están fuera del rango, son añadidos.

- `public void addFirst(T element)`

Clases de equivalencia:

Entrada	Regla	Clase válida	Clase no válida
Elemento	números naturales entre 1 y 5000	1. Rango de $1 \leq \text{elemento} \leq 5000$ 2. Elemento = 1 3. Elemento = 2 4. Elemento = 4999 5. Elemento = 5000	6. Elemento < 1 7. Elemento > 5000 8. Elemento = 5001 9. Elemento = 0 10. Elemento no es número natural

Casos de prueba:

Prueba	Entrada	Clases cubiertas	Salida esperada	Salida
1	elemento = 200	1	Insertado	No insertado
2	elemento = 1	2	Insertado	No insertado
3	elemento = 2	3	Insertado	No insertado
4	elemento = 4999	4	Insertado	No insertado

5	elemento = 5000	5	Insertado	No insertado
6	elemento = -200	6	No insertado	No insertado
7	elemento = 5200	7	No insertado	No insertado
8	elemento = 5001	8	No insertado	No insertado
9	elemento = 0	9	No insertado	No insertado
10	elemento = "abc"	10	No aceptado	No aceptado

Los casos de prueba 1, 2, 3, 4 y 5 no agregan elemento alguno debido a lo que parece ser, una mala implementación del método. Debido a este matiz, podemos afirmar que aquellos elementos fuera de rango que no deberían agregarse, no son agregados.

- `public void addLast(T element)`

Clases de equivalencia:

Entrada	Regla	Clase válida	Clase no válida
Elemento	números naturales entre 1 y 5000	1. Rango de $1 \leq \text{elemento} \leq 5000$ 2. Elemento = 1 3. Elemento = 2 4. Elemento = 4999 5. Elemento = 5000	6. Elemento < 1 7. Elemento > 5000 8. Elemento = 5001 9. Elemento = 0 10. Elemento no es número natural

Casos de prueba:

Prueba	Entrada	Clases cubiertas	Salida esperada	Salida
1	elemento = 200	1	Insertado	Insertado
2	elemento = 1	2	Insertado	Insertado
3	elemento = 2	3	Insertado	Insertado
4	elemento = 4999	4	Insertado	Insertado
5	elemento = 5000	5	Insertado	Insertado
6	elemento = -200	6	No insertado	Insertado

7	elemento = 5200	7	No insertado	Insertado
8	elemento = 5001	8	No insertado	Insertado
9	elemento = 0	9	No insertado	Insertado
10	elemento = "abc"	10	No aceptado	No aceptado

Los casos de prueba 6, 7, 8 y 9 son valores que no deberían de ser añadidos, sin embargo, sí se añaden. Los tests nos arrojan un fallo porque no se cumple con la salida esperada. Por lo demás, el método funciona tal como se espera.

- `public boolean isEmpty()`

Clases de equivalencia:

Entrada	Regla	Clase válida	Clase no válida
Lista	Lista	1. Lista vacía 2. Lista con longitud ≥ 1	

Casos de prueba:

Prueba	Entrada	Clases cubiertas	Salida esperada	Salida
1	Lista vacía	1	True	True
2	Lista con longitud ≥ 1	2	False	False

Este método funciona tal como se tenía previsto.

- `public int size()`

Clases de equivalencia:

Entrada	Regla	Clase válida	Clase no válida
Lista	Lista	1. Lista vacía 2. Lista con longitud ≥ 1	

Casos de prueba:

Prueba	Entrada	Clases cubiertas	Salida esperada	Salida
1	Lista vacía	1	0	0
2	Lista con i elementos	2	i	i

Este método funciona tal como se tenía previsto.

- `public void addNTimes(T element, int n)`

Clases de equivalencia:

Entrada	Regla	Clase válida	Clase no válida
Elemento	Números naturales entre 1 y 5000	1. Rango de $1 \leq \text{elemento} \leq 5000$ 2. Elemento = 1 3. Elemento = 2 4. Elemento = 4999 5. Elemento = 5000	6. Elemento < 1 7. Elemento > 5000 8. Elemento = 5001 9. Elemento = 0 10. Elemento no es número natural
Entero	Números enteros positivos	11. $n \geq 0$ 12. $n = 1$ 13. $n = 0$	14. $n < 0$ 15. $n = -1$ 16. n no es un entero

Casos de prueba:

Prueba	Entrada	Clase cubierta	Salida esperada	Salida
1	element = 200 p = 3	1, 11	Añadir elemento 200 tres veces al final de la lista	Añadir elemento 200 tres veces al final de la lista
2	element = 200 p = 0	1, 13	No se añade nada	IllegalArgument Exception
3	element = 1 p = 1	2, 12	Añadir elemento 1 una vez al final de la lista	Añadir elemento 1 una vez al final de la lista
4	element = 2 p = 1	3, 12	Añadir elemento 2 una vez al final de la lista	Añadir elemento 2 una vez al final de la lista
5	element = 4999 p = 1	4, 12	Añadir elemento 4999 una vez al final de la lista	Añadir elemento 4999 una vez al final de la lista
6	element = 5000 p = 1	5, 12	Añadir elemento 5000 una vez al final de la lista	Añadir elemento 5000 una vez al final de la lista
7	element = -200 p = 1	6, 12	No añadir elemento -200 una vez	Añadir elemento -200 una vez
8	element = 5500 p = 1	7, 12	No añadir elemento 5500 una vez	Añadir elemento 5500 una vez
9	element = 5001 p = 1	8, 12	No añadir elemento 5001 una vez	Añadir elemento 5001 una vez
10	element = 0 p = 1	9, 12	No añadir elemento 0 una vez	Añadir elemento 0 una vez
11	element = "abc" p = 1	10, 12	No aceptado	No aceptado
12	element = 1 p = -10	2, 14	IllegalArgument Exception	IllegalArgument Exception
13	element = 1 p = -1	2, 15	IllegalArgument Exception	IllegalArgument Exception
14	element = 1	2, 16	No aceptado	No aceptado

	p = "abc"			
--	-----------	--	--	--

El método agrega el elemento que se pasa por parámetro **element** es insertado **n** veces al final de la lista, de modo que esa funcionalidad es correcta. Sin embargo, los casos de prueba 7, 8, 9 y 10 producen salidas fallidas, porque se esperaba que no fueran insertados al salirse del rango pero si son insertados. También es importante mencionar que la excepción se lanza para aquellos valores donde n es **menor o igual** que cero.

- `public int indexOf(T elem)`

Clases de equivalencia:

Entrada	Regla	Clase válida	Clase no válida
Elemento	Números naturales entre 1 y 5000	1. Rango de $1 \leq \text{elemento} \leq 5000$ 2. Elemento = 1 3. Elemento = 5000 4. Elemento = 2 5. Elemento = 4999	6. Elemento < 1 7. Elemento > 5000 8. Elemento no es número natural 9. Elemento que no está en la lista. 10. Elemento = 0 11. Elemento = 5001

Casos de prueba:

Prueba	Entrada	Clase cubierta	Salida esperada	Salida
1	elem=200	1	Posición del elemento en la lista	Posición del elemento en la lista: 5
2	elem = 1	2	Posición del elemento en la lista	Posición del elemento en la lista: 1
3	elem = 5000	3	Posición del elemento en la lista	Posición del elemento en la lista: 7
4	elem = 2	4	Posición del elemento en la lista	Posición del elemento en la lista: 2
5	elem = 4999	5	Posición del elemento en la lista	Posición del elemento en la lista: 6

6	elem = 0	10	Devuelve nada	Imposible que devuelva nada
7	elem = 5001	11	Rango no aceptado	NoSuchElementException
8	elem = -200	6	Devuelve nada	Imposible que devuelva nada
9	elem = 5500	7	Rango no aceptado	NoSuchElementException
10	elem = "abc"	8	No aceptado	No aceptado
11	elem=número que no esté en la lista.	9	NoSuchElementException	NoSuchElementException

La lista que se ha utilizado es: (1, 2, 3, 4, 200, 4999,5000).

El método `indexOf` funciona correctamente, devuelve el índice si es que se encuentra el elemento y en caso contrario lanza una excepción, todo esto, ignorando el rango que puede tomar **elem**. Ahora bien, según la documentación, para aquellos elementos menores a 1 la función hace nada, de modo que es imposible porque pase lo que pase devolverá el índice o la excepción.

- `public T removeLast()`

Clases de equivalencia:

Entrada	Regla	Clase válida	Clase no válida
Lista	Lista con longitud ≥ 1	1. Lista con longitud ≥ 1	2. Lista vacía

Casos de prueba:

Prueba	Entrada	Clases cubiertas	Salida esperada	Salida
1	Lista con longitud ≥ 1	1	Último elemento eliminado	Último elemento eliminado
2	Lista vacía	2	<code>singleList.Empty</code>	<code>singleList.Empty</code>

			tyCollectionEx ception	tyCollectionEx ception
--	--	--	---------------------------	---------------------------

Este método funciona tal como se tenía previsto.

El caso de prueba 2 se realiza mediante el método `removeLastElement_2()`, el cual lanza una excepción debido a que se quiere eliminar un elemento de una lista en la que no hay elementos. Por lo tanto, hemos creado el método `excRemoveLastElement_2()` para que se encargue de tratar dicha excepción.

- `public T removeLast(T elem)`

Clases de equivalencia:

Entrada	Regla	Clase válida	Clase no válida
Elemento	números naturales entre 1 y 5000	1. Rango de $1 \leq \text{elemento} \leq 5000$ 2. Elemento = 1 3. Elemento = 2 4. Elemento = 4999 5. Elemento = 5000	6. Elemento < 1 7. Elemento > 5000 8. Elemento = 5001 9. Elemento = 0 10. Elemento no es número natural 11. Lista vacía 12. Lista sin el elemento

Casos de prueba:

Prueba	Entrada	Clase cubierta	Salida esperada	Salida
1	elem = 200	1	200	null y borra siguiente elemento a 200
2	elem = "abc"	10	No aceptado	No aceptado
3	elem = 10000	7	No aceptado	null
4	elem = 1	2	1	null y borra siguiente elemento a 1
5	elem = 2	3	2	null y borra siguiente elemento a 2

6	elem = 4999	4	4999	null y borra siguiente elemento a 4999
7	elem = 5000	5	5000	null y borra siguiente elemento a 5000
8	elem = 0	9, 6	No aceptado	null
9	elem = 5001	8	No aceptado	null
10	Lista vacía	11	singleList.EmptyCollectionException	singleList.EmptyCollectionException
11	Lista sin el elemento a buscar	12	java.util.NoSuchElementException	null

La lista con la que hemos probado este método es (1, 2, 200, 4999, 5000, 1, 2), la cual consta de dos elementos repetidos. Esto lo hemos hecho aposta para probar que si queremos eliminar la segunda ocurrencia del número 1, nos elimine dicha ocurrencia y no la primera.

De todas formas, esto da igual ya que el método `T removeLast(T elem)` está mal implementado, ya que lo que hace es borrar el elemento siguiente al que le pasamos por parámetro (p. ej. si queremos borrar el 2 nos borra el 200 en la lista anterior) y nos devuelve un null siempre en vez del número borrado.

***NOTA:** Ambos métodos **RemoveLast** nos obligan a implementar el lanzamiento de la excepciones con **throws** en la declaración de los métodos de prueba aún incluso no necesitando tratarlas porque estas no se llegan a producir.

- `public AbstractSingleLinkedListImpl<T> reverse()`

Clases de equivalencia:

Entrada	Regla	Clase válida	Clase no válida
Lista	Lista	<ol style="list-style-type: none"> 1. Lista vacía 2. Lista con longitud ≥ 1 	

Casos de prueba:

Prueba	Entrada	Clases cubiertas	Salida esperada	Salida
1	Lista vacía	1	Lista vacía	Lista vacía
2	Lista con i elementos	2	Una lista invertida de los elementos	Una lista invertida de los elementos

Este método funciona tal como se tenía previsto.

- `public int isSubList(AbstractSingleLinkedListImpl<T> part)`

Clases de equivalencia:

Entrada	Regla	Clase válida	Clase no válida
Lista	Lista	1. Lista vacía. 2. Lista con longitud ≥ 1 .	

Casos de prueba:

Prueba	Entrada	Clases cubiertas	Salida esperada	Salida
1	Lista vacía	1	1	1
2	Sublista no existente	2	-1	-1
3	Sublista con elementos iguales a partir del elemento i	2	Posición a partir de la cual se encuentra la sublista.	Posición a partir de la cual se encuentra la sublista.

Este método funciona tal como se tenía previsto.

- `public T getAtPos(int pos)`

Clases de equivalencia:

Entrada	Regla	Clase válida	Clase no válida
Pos	$\text{pos} \geq 1$	<ol style="list-style-type: none">1. Número entero positivo2. $\text{pos} = 1$3. $\text{pos} = 2$	<ol style="list-style-type: none">4. Número entero negativo5. $\text{pos} = 0$6. Posición mayor a la lista7. Posición no es un número entero positivo

Casos de prueba:

Prueba	Entrada	Clases cubiertas	Salida esperada	Salida
1	$\text{pos} = 5$	1	Elemento de la posición 5	Elemento de la posición 5
2	$\text{pos} = 1$	2	Elemento de la posición 1	Elemento de la posición 1
3	$\text{pos} = 2$	3	Elemento de la posición 2	Elemento de la posición 2
4	$\text{pos} = -5$	4	IllegalException	IllegalException
5	$\text{pos} = 0$	5	IllegalException	IllegalException
6	$\text{pos} = 100$ (size = 10)	6	IllegalException	IllegalException
7	$\text{pos} = \text{"abc"}$	7	No aceptado	No aceptado

Este método funciona tal como se tenía previsto.

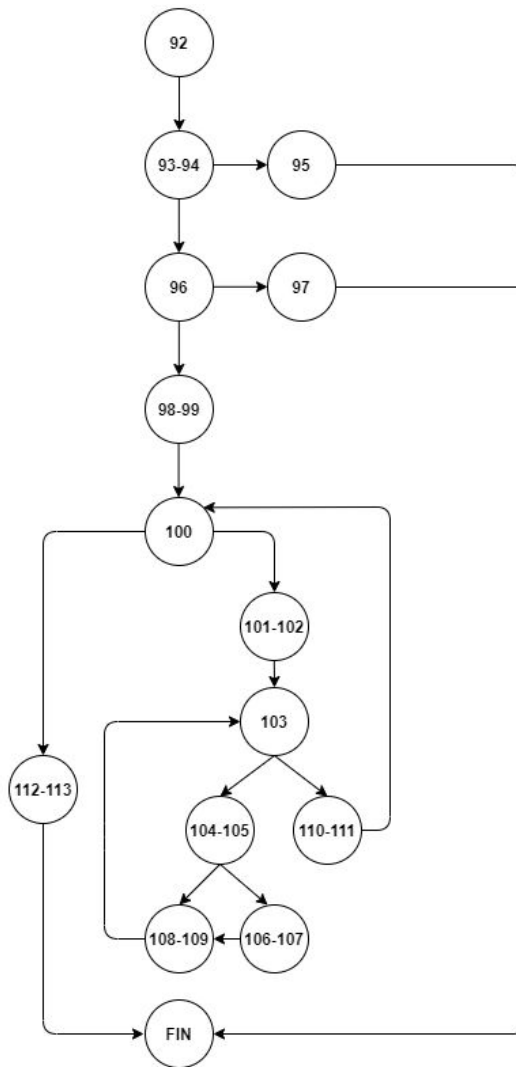
Test de caja blanca

En la caja negra hemos utilizado la técnica de *Path-Testing* , es decir, un caso de prueba para cada camino independiente sacado del grafo normalizado.

- `public int numPalabras(int inicio, int fin, String palabra);`

```
92 public int numPalabras(int inicio, int fin, String palabra) {
93     int valor = inicio;
94     if (valor <= 0)
95         throw new IllegalArgumentException("La línea de inicio no puede ser menor o igual a cero");
96     if (fin > this.editor.size())
97         throw new IllegalArgumentException("La línea fin no puede ser mayor que el máximo de líneas");
98     int apariciones = 0;
99
100     while (!editor.isEmpty() && valor < fin) {
101         this.lista = this.editor.getAtPos(valor);
102         int pos = 1;
103         while (pos <= this.lista.size()) {
104             String cadena = this.lista.getAtPos(pos);
105             if (cadena.equals(palabra)) {
106                 apariciones++;
107             }
108             pos++;
109         }
110         valor++;
111     }
112     return apariciones;
113 }
```

Grafo normalizado:



A partir del grafo se puede obtener la complejidad ciclomática y el número de caminos independientes:

- Complejidad : $N^{\circ} \text{ aristas} - N^{\circ} \text{ nodos} + 2 = 19 - 15 + 2 = 6$
- Número de caminos independientes: 6

Casos de prueba:

1. $92 \rightarrow 93,94 \rightarrow 95 \rightarrow \text{FIN}$

Si el inicio es menor o igual a 0, lanza una excepción.

2. $92 \rightarrow 93,94 \rightarrow 96 \rightarrow 97 \rightarrow \text{FIN}$

Si el fin es mayor al tamaño del editor, lanza una excepción.

3. $92 \rightarrow 93,94 \rightarrow 96 \rightarrow 98,99 \rightarrow 100 \rightarrow 112,113 \rightarrow \text{FIN}$

Si el editor está vacío, devuelve 0.

4. 92 → 93,94 → 96 → 98,99 → 100 → 101,102 → 103 → 110,111 → 100 → 112,113 → FIN

Si el fichero no está vacío pero una de las líneas tiene longitud 0, no entra al bucle while interno.

5. 92 → 93,94 → 96 → 98,99 → 100 → 101,102 → 103 → 104,105 → 108,109 → 103 → 110,111 → 100 → 112,113 → FIN

Si el fichero no está vacío y la palabra no está presente, no entra a la condición if.

6. 92 → 93,94 → 96 → 98,99 → 100 → 101,102 → 103 → 104,105 → 106, 107 → 108,109 → 103 → 110,111 → 100 → 112,113 → FIN

Si el fichero no está vacío y la palabra está presente, entra a la condición if.

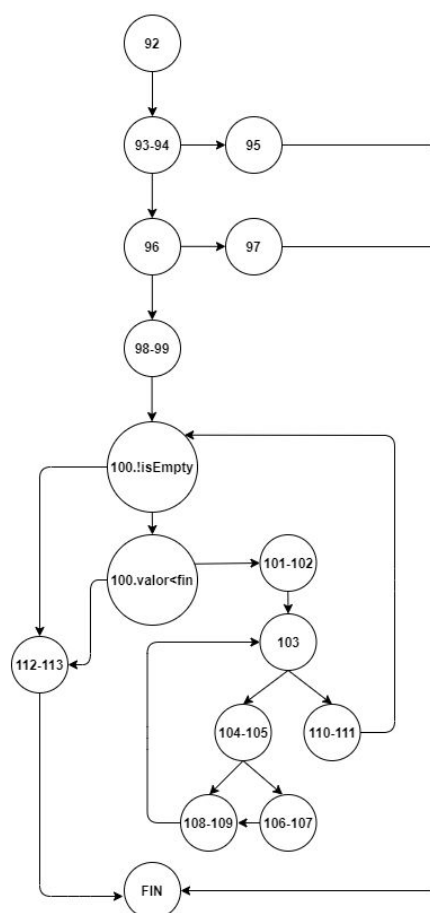
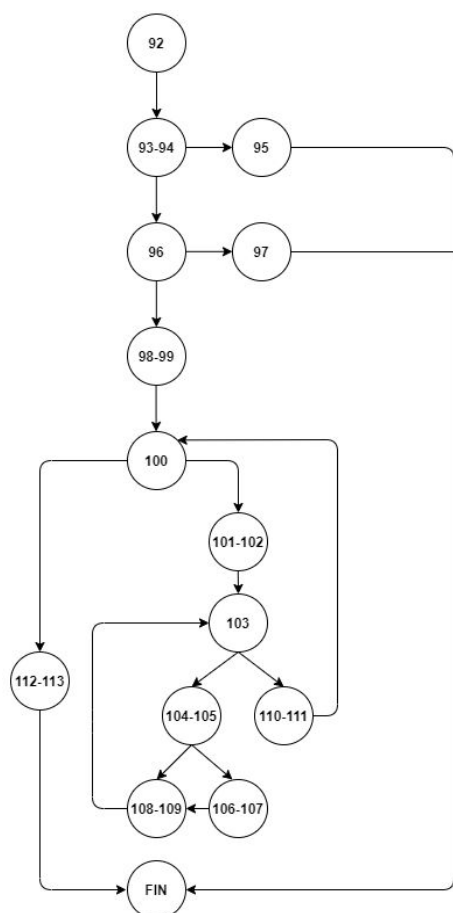
Condiciones múltiples:

El nodo 100 tiene condiciones múltiples.

```
100      while (!editor.isEmpty() && valor < fin)
```

Condiciones				
!editor.isEmpty()	T	T	F	F
valor < fin	T	F	T	F
Acciones				
Entrar bucle	X			
No entrar bucle		X	X	X

Grafo con el nodo complejo y con el nodo expandido respectivamente:



	Grafo con nodo complejo	Grafo con nodo expandido
Complejidad ciclomática	6	7
Número de caminos independientes	6	7

Si bien en este apartado, se busca cubrir todos los caminos posibles, nos hemos encontrado con un problema y es que el método **numPalabras** no cuenta las palabras presentes en la última línea. Este error se encontró al intentar realizar solo un bucle por cada caso de prueba, al haber tan solo una línea, dicha línea no era leída. Por último, el método lanza las excepciones cuando se requiere de forma correcta y se cuenta la aparición de una palabra en el fichero correctamente siempre y cuando no aparezca en la última línea.

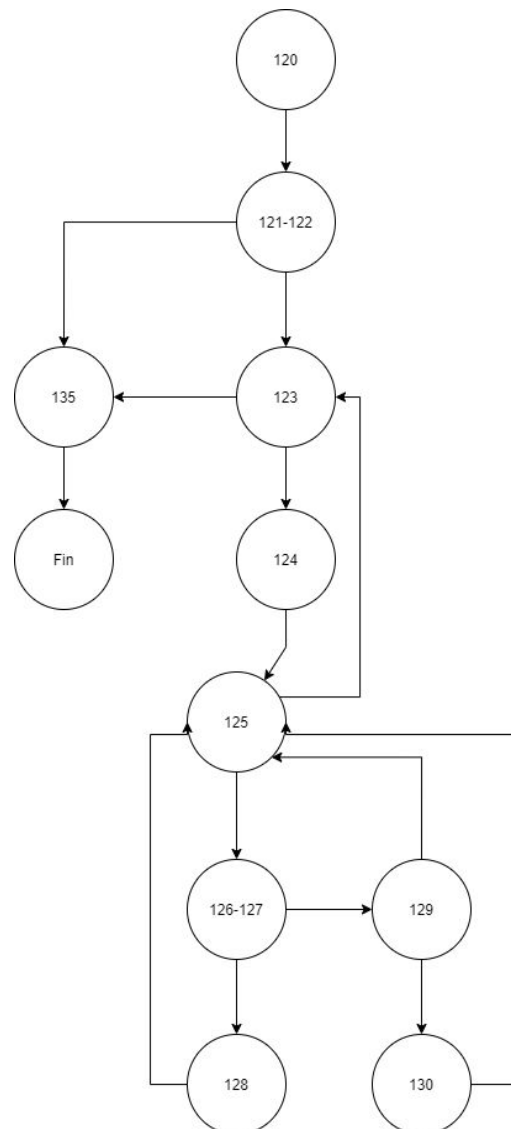
- **public String mayorLongitud():**

```

120 public String mayorLongitud() {
121     String mayor = null;
122     if (this.editor.size() > 0) {
123         for (int i = 1; i <= this.editor.size(); i++) {
124             this.lista = this.editor.getAtPos(i);
125             for (int pos = 1; pos <= this.lista.size(); pos++) {
126                 String cadena = this.lista.getAtPos(pos);
127                 if (mayor == null) {
128                     mayor = cadena;
129                 } else if (cadena.length() > mayor.length()) {
130                     mayor = cadena;
131                 }
132             }
133         }
134     }
135     return mayor;
136 }

```

Grafo normalizado:



A partir del grafo se puede obtener la complejidad ciclomática y el número de caminos independientes:

- Complejidad : N° de arista - N° del nodo + 2 = 15 - 11 + 2 = 6
- Número de caminos independientes: 6

Casos de prueba:

1. 120 → 121,122 → 135 → Fin

Si el fichero txt está vacío, termina la función.

2. 120 → 121,122 → 123 → 135 → Fin

Si el fichero txt no está vacío y la longitud del fichero es 0, termina.

El caso 2 es imposible que suceda, ya que si completa la condición de `editor.size()` es mayor que 0, entonces al menos va a tener un tamaño de 1.

3. 120 → 121,122 → 123 → 124 → 125 → 123 → 135 → Fin

Si el fichero txt no está vacío, la longitud del fichero no es 0 y la longitud de la lista es 0, termina.

El caso 3 es imposible que suceda, ya que si el fichero no está vacío, va a tener al menos una cadena y la condición del tamaño de la lista nunca va a ser menor que 1.

4. 120 → 121,122 → 123 → 124 → 125 → 126,127 → 128 → 125 → 123 → 135 → Fin

Si el fichero txt no está vacío, la longitud del fichero no es 0, la longitud de la lista no es 0 y se coge el primer elemento, termina.

5. 120 → 121,122 → 123 → 124 → 125 → 126,127 → 129 → 125 → 123 → 135 → Fin

Si el fichero txt no está vacío, la longitud del fichero no es 0, la longitud de la lista no es 0 y el actual elemento es menor que el anterior, termina.

6. 120 → 121,122 → 123 → 124 → 125 → 126,127 → 129 → 130 → 125 → 123 → 135 → Fin

Si el fichero txt no está vacío, la longitud del fichero no es 0, la longitud de la lista no es 0 y el actual es mayor que el anterior, cambiar anterior por actual, termina.

Condiciones múltiples:

No tiene condiciones múltiples.

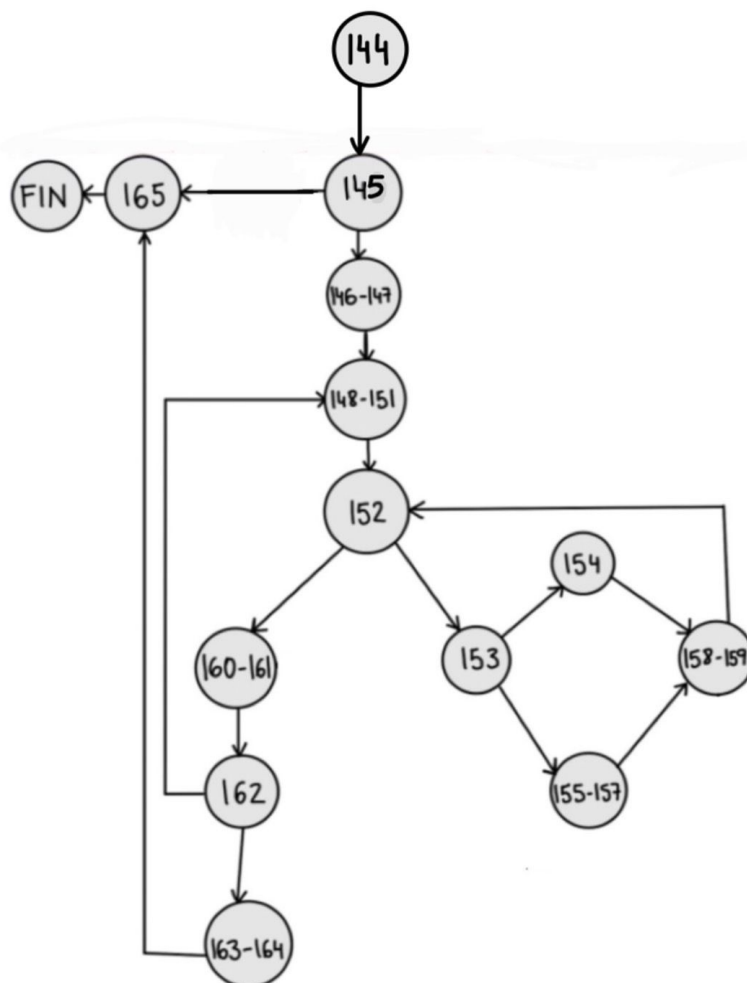
- **public void** **sustituirPalabra** (**String** palabra, **String** nuevaPalabra);

```

144 public void sustituirPalabra(String palabra, String nuevaPalabra) {
145     if (this.editor.size() > 0) {
146         AbstractSingleLinkedListImpl<AbstractSingleLinkedListImpl<String>> nuevoEditor = new SingleLinkedListImpl
147         int i = 1;
148         do {
149             AbstractSingleLinkedListImpl<String> aux = new SingleLinkedListImpl<String>();
150             this.lista = this.editor.getAtPos(i);
151             int j = 1;
152             while (j <= this.lista.size()) {
153                 if (this.lista.getAtPos(j).equals(palabra)) {
154                     aux.addLast(nuevaPalabra);
155                 } else {
156                     aux.addLast(this.lista.getAtPos(j));
157                 }
158                 j++;
159             }
160             nuevoEditor.addLast(aux);
161             i++;
162         } while (i <= this.editor.size());
163         editor = nuevoEditor;
164     }
165 }

```

Grafo normalizado:



- Complejidad : N° de aristas - N° de nodos + 2 = 17 - 14 + 2 = 5
- Número de caminos independientes: 5

Casos de prueba:

1. 144 → 145 → 165 → FIN

Como el tamaño del *editor* es igual a 0, se termina la ejecución directamente.

2. 144 → 145 → 146,147 → 148,149,150,151 → 152 → 160,161 → 162 → 163,164 → 165 → FIN

El tamaño del fichero *editor* es mayor que 0, por lo que nos metemos en el bucle *do-while* pero como el tamaño de lista es 0 no entramos en el *while*.

Cuando termina el *do-while* no volvemos a entrar porque *i* es menor que el tamaño de *editor*.

3. 144 → 145 → 146,147 → 148,149,150,151 → 152 → 160,161 → 162 → 148,149,150,151 → 152 → 160,161 → 162 → 163,164 → 165 → FIN

Igual que el caso anterior pero al comprobar la condición del *do-while* se vuelve a meter en el bucle. En este caso pasamos por la arista que une a los nodos 162 con 148.

4. 144 → 145 → 146,147 → 148,149,150,151 → 152 → 153 → 154 → 158,159 → 152 → 160,161 → 162 → 163,164 → 165 → FIN

Se mete en el *do-while* y en el *while*, ya que el tamaño de la lista es mayor que la variable de control *j*. Como la palabra de la lista es igual a la que se compara con la palabra pasado por parámetro, se añade al final de la lista *aux* la nueva palabra. Al aumentar *j* al final del bucle, se sale del *while* y por la misma razón pero con la variable *i*, se sale del *do-while*.

5. 144 → 145 → 146,147 → 148,149,150,151 → 152 → 153 → 155,156,157 → 158,159 → 152 → 160,161 → 162 → 163,164 → 165 → FIN






Se mete en el *do-while* y en el *while*, ya que el tamaño de la lista es mayor que la variable de control *j*. Como la palabra de la lista es diferente a la que se compara con la palabra pasado por parámetro, se añade la palabra leída al final de la lista *aux*. Al aumentar *j* al final del bucle, se sale del *while* y por la misma razón pero con la variable *i*, se sale del *do-while*.

Condiciones múltiples:

No tiene condiciones múltiples.

Informe de cobertura final

En la siguiente imagen se muestra el informe de cobertura final. Como se puede ver, para los tests de caja negra se ha obtenido una cobertura del 84,5 % mientras que para los tests de caja blanca se ha obtenido una cobertura del 94,6 %, superando el 80 % solicitado como requisito.

Element	Coverage
▼ editText	 85,4 %
▼ src/test/java	 86,4 %
▼ singlelist.edittext	 86,4 %
> SingleLinkedListTest.java	 84,5 %
> EditorTest.java	 94,6 %