PRÁCTICA DISEÑO

EJERCICIO 2

GRUPO 23-03

Diego Suárez Ramos

Ander Varela

Introducción:

En una empresa gestionan los proyectos dividiéndolos en sus tareas correspondientes. Las tareas tienen dependencias entre sí, que se incluyen en un documento de texto. Quién decide el orden de ejecución de las tareas es un gestor de proyectos, que según las condiciones del entorno decidirá cual es el mejor orden de ejecución de las mismas. Desarrolla una solución, basada en principios y patrones de diseño, que nos permita fácilmente almacenar el gráfico de tareas y obtener los distintos ´ordenes de ejecución de tareas descritos.

Implementación:

Tenemos una clase "LeerArchivo" que nos lee las tareas de un archivo, esta clase es llamada por la clase "Tareas" para exportar las tareas de un archivo y almacenarlas en forma de grafo y decidir en que orden se realizarán, para eso hace uso de alguna de las distintas implementaciones de la interfaz "OrdenRealizacionTareas" para decidir el orden de hacer las tareas.

PRINCIPIOS USADOS:

-Responsabilidad única:

Cada objeto debe tener una responsabilidad única que esté enteramente encapsulada en la clase. Todos los servicios que provee el objeto están estrechamente alineados con dicha responsabilidad.

Intentamos hacer un diseño donde cada clase realiza únicamente una tarea. De esta forma, tenemos que, "LeerArchivo" solo se encarga de leer un fichero; "Tareas" se encarga de gestionar el grafo de tareas que tenemos y el orden en que se realizarán; y las diferentes clases que extienden a la clase abstracta "OrdenRealizacionTareas" se encargan de, según su implementación, decidir el orden en el que se harán las tareas.

-Principio de abierto-cerrado:

Las entidades software (clases, módulos, etc.) deberían ser abiertas para permitir su extensión, pero cerradas frente a la modificación.

Nuestro sistema es extensible, porque permite agregar nuevas tareas al grado de tareas para realizar y además, podemos crear nuevas formas para decidir el orden de realización de las tareas extendiendo a "OrdenRealizacionTareas" . Así mismo, es cerrado, porque en ningún momento se muestra la implementación de las clases, todo está definido como privado y no se puede modificar ningún atributo de ninguna clase, ya que todos están declaradas como final, por lo que está cerrada la modificación, a excepción de la clase abstracta "OrdenRealizacionTareas" que tiene los atributos como protected, para que puedan ser modificados y usados por sus subclases a fin de tener diferentes algoritmos de decidir el orden de las tareas. Además, en "Tareas" se usa como interfaz externa las interfaces List y Map de la API de Java(visibles para el cliente) , pero no son visibles para un cliente las clases que se usan de manera interna, en este caso, son ArrayList y TreeMap respectivamente.

-Principio de sustitución de Liskov:

Las clases derivadas deben ser sustituibles por sus clases base. Aquellos métodos que utilicen referencias a clases base deben ser capaces de usar objetos de clases derivadas sin saberlo.

Nuestra interfaz "OrdenRealizacion" puede ser sustituida por cualquier clase que la implemente, debido a que no incluimos ninguna restricción en ningún método, además el funcionamiento interno de las clases es muy parecido. Por ejemplo, cuando usamos el método de realizar, de la clase "Tareas", lo que se pasa como parámetro es un "OrdenRealizacion", y se usa de manera externa la interfaz genérica de éste, sin saber la implementación interna que tiene, por lo tanto, puedes pasar cualquier clase que implemente a "OrdenRealizacion", que el método será capaz de usarlo sin saber de que clase derivada se trata.

-Principio de inversión de la dependencia:

Depende de abstracciones y no de concreciones.

Cuando usamos el método de realizar, de la clase "Tareas", en vez de depender una clase concreta para decidir el orden de las tareas, va depender de la clase abstracta "OrdenRealizacionTareas". De esta forma, tanto nosotros como cualquier otra persona que quiera añadir nuevas formas de ordenar las tareas, podrá hacerlo, sin comprometer el funcionamiento del sistema y sin tener que modificar nada de la clase "Tareas".

Aparte de los principios SOLID aplicamos los siguientes principios:

-Principio encapsula lo que varía:

Identifica los aspectos de tu aplicación que varían y sepáralos de aquellos que permanecen estables

Sabemos que nuestro sistema va a permanecer estable, exceptuando las partes que dan lugar a cambio. El método de introducir las tareas, como el de manejar el grafo fueron diseñados para que no se tengan que modificar y duren en el tiempo. Mientras que la parte que sí se puede cambiar son en las que se usen nuevas formas de decidir el orden a realizar las tareas (las clases que extiendan a "OrdenRealizarTarea"), ya que siempre podemos añadir más.

PATRONES:

En un principio habíamos escogido el patrón Estrategia, pero al observar que teníamos dos partes bien diferenciadas en todos los algoritmos decidimos hacer una implementación del patrón método plantilla

-Patrón Método Plantilla:

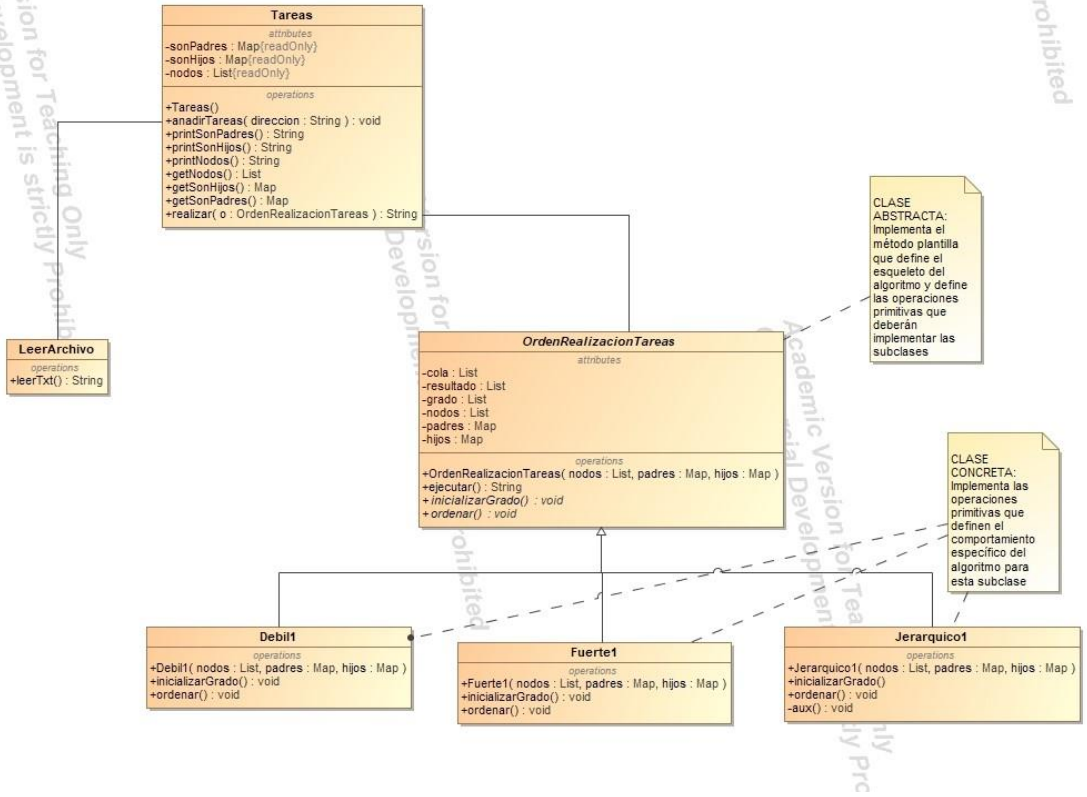
Define el esqueleto de un algoritmo en una operación pero difiriendo alguno de los pasos a las subclases.

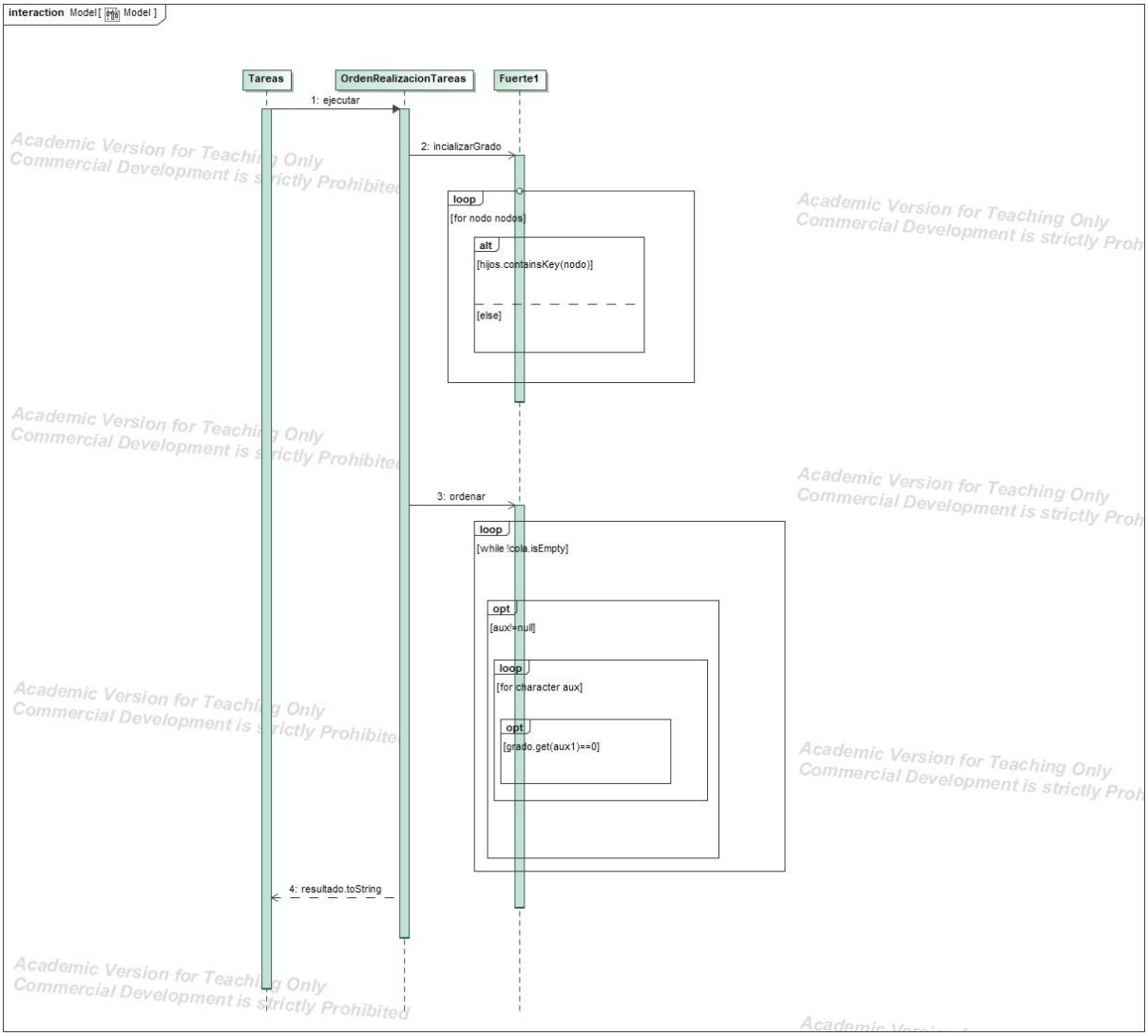
En nuestra aplicación nos podemos fijar, que tenemos una familia de algoritmos que vienen siendo los distintos algoritmos para decidir en qué orden realizar las tareas, según las dependencias. Inicialmente pensamos en utilizar el patrón Estrategia, pero una vez se observa la implementación de los distintos algoritmos de ordenación nos damos cuenta de que se diferencian 2 partes bien marcadas, la parte de "inicializarGrado" y la parte de "ordenar". Por lo tanto, se adaptó el código para que se haga uso del método plantilla y así separar las dos partes, aunque quede algo forzado. Podemos diferenciar las siguientes partes del método plantilla:

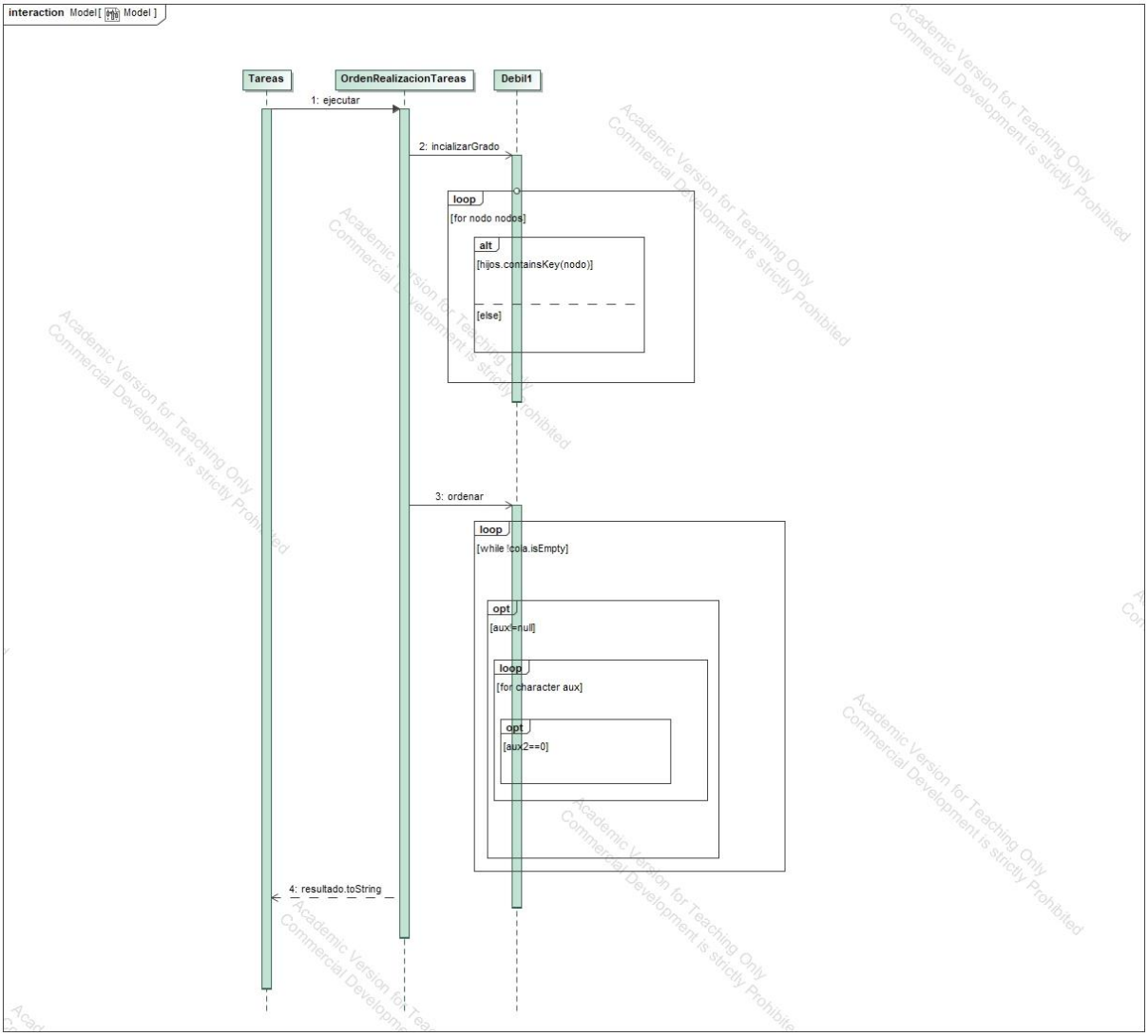
ClaseAbstracta: La clase abstracta "OrdenRealizacionTareas" desempeña este rol. Implementa el método plantilla que define el esqueleto del algoritmo (método "ejecutar") y define las operaciones primitivas que deberán implementar las subclases (métodos "ordenar" y "inicializarGrado").

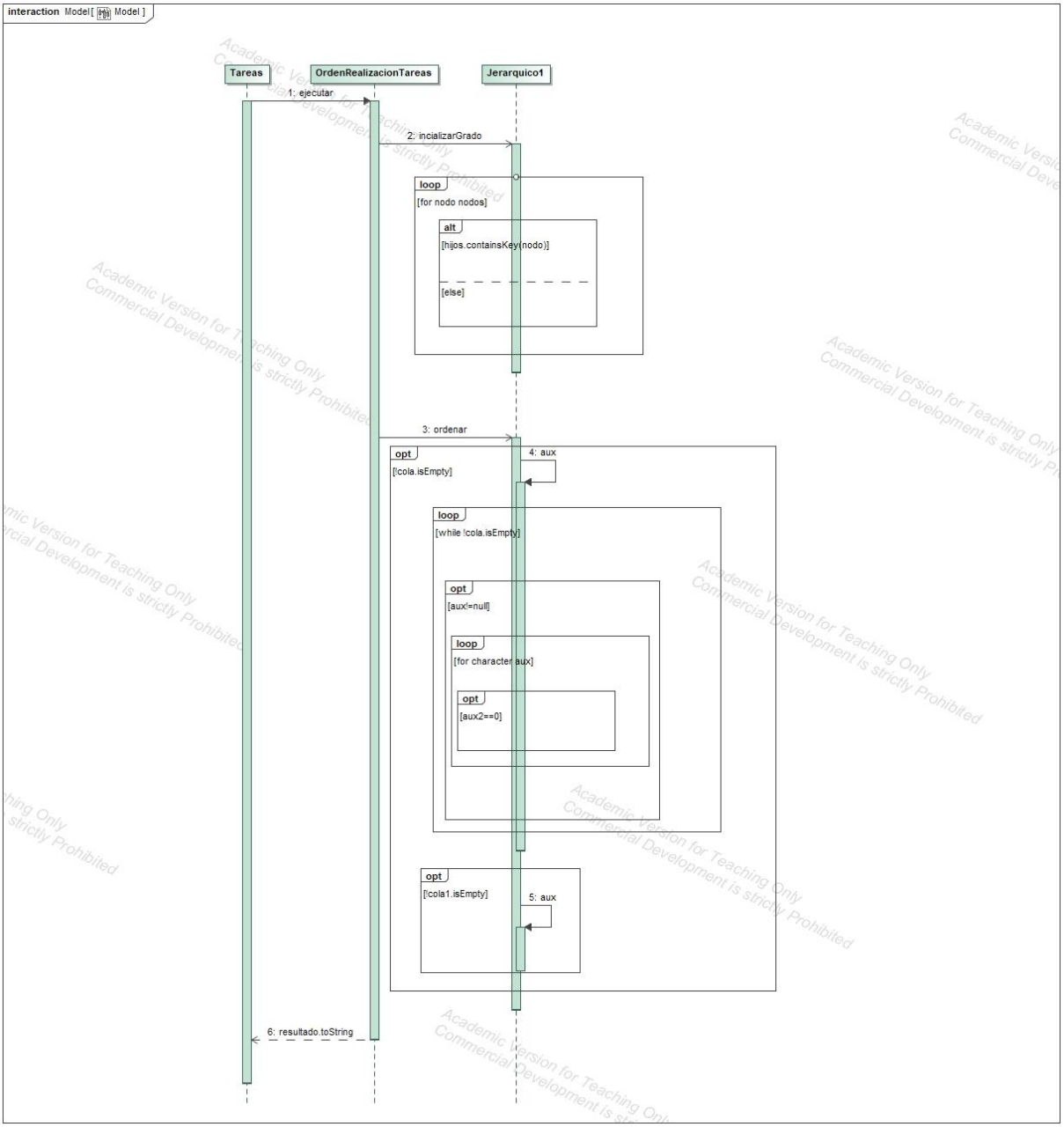
ClaseConcreta: Las clases "Debil1", "Fuerte1", "Jerarquico1" desarrollan este rol, implementan las operaciones primitivas que definen el comportamiento específico del algoritmo para cada una de estas subclases. Son las clases que implementan "ordenar" y "inicializarGrado"

A continuación se muestran tanto el diagrama de clases como los diagramas dinámicos de comunicación(uno para cada implementación de elegir el orden de las tareas, Debil1, Fuerte1, Jerarquico1


Tareas
attributes
-sonPadres : Map{readOnly}
-sonHijos : Map{readOnly}
-nodos : List{readOnly}
operations
+Tareas()
+anadirTareas(direccion : String) : void
+printSonPadres() : String
+printSonHijos() : String
+printNodos() : String
+getNodos() : List
+getSonHijos() : Map
+getSonPadres() : Map
+realizar(o : OrdenRealizacionTareas) : String
LeerArchivo
operations
+leerTxt() : String
OrdenRealizacionTareas
attributes
-cola : List
-resultado : List
-grado : List
-nodos : List
-padres : Map
-hijos : Map
operations
+OrdenRealizacionTareas(nodos : List, padres : Map, hijos : Map)
+ejecutar() : String
+ inicializarGrado() : void
+ ordenar() : void
CLASE ABSTRACTA: Implementa el método plantilla que define el esqueleto del algoritmo y define las operaciones primitivas que deberán implementar las subclases
CLASE CONCRETA: Implementa las operaciones primitivas que definen el comportamiento específico del algoritmo para esta subclase
Debil1
operations
+Debil1(nodos : List, padres : Map, hijos : Map)
+inicializarGrado() : void
+ordenar() : void
Fuerte1
operations
+Fuerte1(nodos : List, padres : Map, hijos : Map)
+inicializarGrado() : void
+ordenar() : void
Jerarquico1
operations
+Jerarquico1(nodos : List, padres : Map, hijos : Map)
+inicializarGrado()
+ordenar() : void
-aux() : void


interaction Model [Model]
Tareas
OrdenRealizacionTareas
Fuerte1
1: ejecutar
2: incializarGrado
loop
[for nodo nodos]
alt
[hijos.containsKey(nodo)]
[else]
3: ordenar
loop
[while !cola.isEmpty]
opt
[aux!=null]
loop
[for character aux]
opt
[grado.get(aux1)==0]
4: resultado.toString
Academic Version for Teaching Only
Commercial Development is strictly Prohibited


interaction Model [Model]
Tareas
OrdenRealizacionTareas
Debil1
1: ejecutar
2: incializarGrado
loop
[for nodo nodos]
alt
[hijos.containsKey(nodo)]
[else]
3: ordenar
loop
[while !cola.isEmpty]
opt
[aux!=null]
loop
[for character aux]
opt
[aux2==0]
4: resultado.toString


interaction Model[Model]
Tareas
OrdenRealizacionTareas
Jerarquico1
1: ejecutar
2: incializarGrado
loop
[for nodo nodos]
alt
[hijos.containsKey(nodo)]
[else]
3: ordenar
opt
[!cola.isEmpty]
4: aux
loop
[while !cola.isEmpty]
opt
[aux!=null]
loop
[for character aux]
opt
[aux2==0]
opt
[!cola1.isEmpty]
5: aux
6: resultado.toString