Análisis y diseño de software

dit
UPM

29.01.2019

# Ejercicio 1

Juan Antonio de la Puente <jpuente@dit.upm.es>



ETSIT UPM

# Objetivos

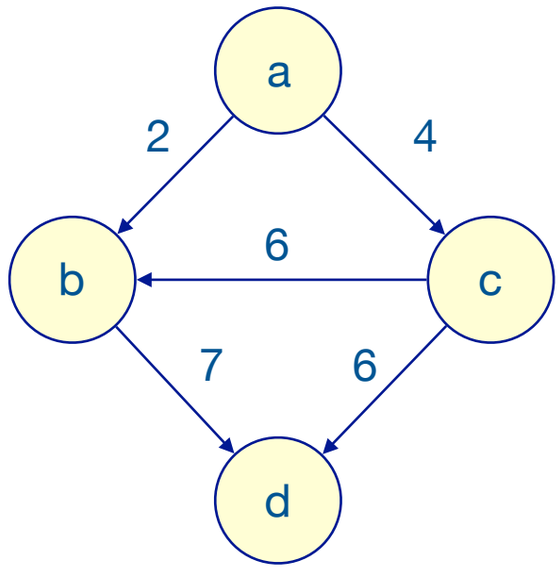
- Programar y probar algoritmos sobre grafos
  - Algoritmo de Dijkstra: camino mínimo

- Tareas:
  1. Buscar el algoritmo en Internet
  2. Adaptar el código al marco del enunciado

- Especificación: javadoc
- Pruebas unitarias: JUnit4

# Grafos

- Un grafo es una estructura de nodos unidos por arcos
- Consideraremos grafos con arcos orientados y pesos enteros
- Ejemplo




Nodos

a, b, c, d

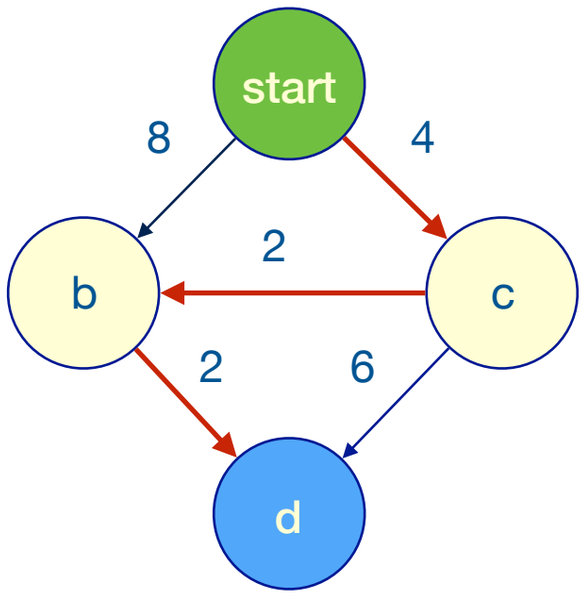
Arcos

(a,b,2)
(a,c,4)
(c,b,6)
(c,d,6)
(b,d,7)

https://en.wikipedia.org/wiki/Directed_graph

# Algoritmo de Dijkstra

- Calcular el camino más corto desde un nodo origen a un destino
- Ejemplo




distancias mínimas

| | | |
|---|---|---|
| **b** | [b,c] | 6 |
| **c** | [c] | 4 |
| **d** | [c,b,d] | 8 |

- Se van visitando todos los nodos y actualizando las distancias más cortas desde el origen

  https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

# Clases

- Node.java
  - NodeTest.java
- Link.java
  - LinkTest.java
- Graph.java
  - GraphTest.java
- Dijkstra.java
  - DijkstraTest.java

Se proporciona

- la especificación (javadoc) de las clases
- ejemplos de pruebas básicas (completar)

# Desarrollo basado en pruebas

- Escribir el esqueleto de la clase
  - a partir de la especificación
- Escribir una batería de pruebas unitarias con JUnit
  - casos normales, casos límite, casos de error
- Escribir código y probar
  - corregir hasta pasar las pruebas
- Refactorizar para mejorar
  - transformaciones del código que no modifican el comportamiento
    - rename, inline, extract variable, extract method
  - pasan las mismas pruebas
  - más legible
  - más fácil de mantener

# Ejemplo: node

es.upm.dit.adsw.ej1

## Class Node

java.lang.Object
    es.upm.dit.adsw.ej1.Node

---

```
public class Node
extends java.lang.Object
```

Nodes in a graph.

### *Constructor Summary*

**Constructors**

| Constructor and Description |
| --- |
| `Node(java.lang.String name, int x, int y)`<br>Constructor. |

### *Method Summary*

**All Methods** | **Instance Methods** | **Concrete Methods**

| Modifier and Type | Method and Description |
| --- | --- |
| `java.lang.String` | `getName()`<br>Getter. |
| `int` | `getX()`<br>Getter. |
| `int` | `getY()`<br>Getter. |
| `java.lang.String` | `toString()`<br>String representation. |

**Methods inherited from class java.lang.Object**

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait`

# Esqueleto de la clase (java)

```java
package es.upm.dit.adsw.ej1;

/**
 * Nodes in a graph.
 * @author Juan A. de la Puente
 * @version 2019.01.27
 */
public class Node {
    // atributes

    /**
     * Constructor.
     * @throws IllegalArgumentException if name is null or empty.
     */
    public Node(String name, int x, int y) {}

    /**
     * String representation.
     */
    @Override
    public String toString() {}

    /**
     * Getter.
     */
    public String getName() {}

    /**
     * Getter.
     */
    public int getX() {}

    /**
     * Getter.
     */
    public int getY() {}
}
```

# Pruebas unitarias (JUnit4)

```java
package es.upm.dit.adsw.ej1;

import org.junit.Test;
import static org.junit.Assert.*;

/**
 * pruebas de la clase Node.
 * @author Juan A. de la Puente
 * @version 2019.01.27
 */
public class NodeTest {

    @Test
    public void test01 () {
        Node node = new Node("a", 0, 0);
        assertEquals("a", node.getName());
        assertEquals(0, node.getX());
        assertEquals(0, node.getY());
        assertEquals("a", node.toString());
    }

    @Test (expected = IllegalArgumentException.class)
    public void test02() {
        Node node = new Node("", 0, 0);
    }

    @Test (expected = IllegalArgumentException.class)
    public void test03() {
        Node node = new Node(null, 0, 0);
    }
}
```

# Otras clases

- Link

```
Link(String src, String dst, int weight)
...
```

- Graph

```
Graph()
// lista de nodos, lista de arcos
addNode(Node node)
addLink(Link link)
...
```

Implementar y probar

# Algoritmo de Dijkstra

- Buscar algoritmo en internet
  - opción 1: busque el algoritmo explicado en lenguaje natural (ej. en español)
    - y prográmelo
    - probablemente necesite usar java.util.List<> y java.util.Map<>
  - opción 2: busque el algoritmo resuelto en java
    - tendrá que envolver el algoritmo adaptando sus estructuras de datos de Graph a las de la solución encontrada

# Pruebas

- Venga de donde venga la solución, ¡pruébela!
- si esta mal, la responsabilidad será del alumno
- probablemente
  - los casos singulares (situaciones límite) los tenga que diseñar
  - para casos normales puede recurrir a los mil ejemplos resueltos en Internet
  - en todo caso debe evitar escenarios en los que haya varias soluciones correctas
    - las pruebas se vuelven más complejas sin aportar más seguridad al código
- se proporcionan ejemplos
  - hay que completarlos

# Realización del ejercicio

- Se pueden formar grupos de 1, 2 o 3 alumnos
  - obligatorio poner los nombres en @author en cada fichero java
- Si el grupo es de 2 o más alumnos, conviene repartir tareas
  - al menos deben ser diferentes personas las que preparen los casos de prueba y las que preparen la solución

# Cómo proceder

- Descargue el archivo ADSW_ejercicio1.zip de moodle e importe el proyecto que contiene en eclipse
- Implemente las clases que se piden en la carpeta src
  - paquete es.upm.dit.adsw.ej1
  - respete la especificación de javadoc
- Ejecute las pruebas hasta que esté todo bien
- Ejecute `Entrega.launch` desde eclipse
  - obtendrá una nota provisional y mensajes orientativos
  - la nota definitiva se asigna por los profesores
  - puede comprobar que el fichero entrega.zip se ha subido a moodle

# Criterios de evaluación

- Corrección del código
- Cobertura de las pruebas
- Estilo de programación