Programación II

Ejercicio Práctico Opcional: Taller de Reparaciones

Dpto. LSIIS. Unidad de Programación

Objetivo: El objetivo de esta práctica es la familiarización del alumno con la programación orientada a objetos en Java y el uso de TADs clásicos.

Evaluación: La práctica se podrá realizar en grupos de dos alumnos o de forma individual. La nota que se obtenga en esta práctica en ningún caso bajará la nota de las prácticas de la asignatura (NP). Por el contrario, en caso de que el alumno apruebe esta práctica con una nota NTaller, esta nota servirá como bonificación de forma que la nota de prácticas del alumno con la bonificación (NP') se calculará así NP' = NP + 1.5*NTaller/10. Se mantendrá el mismo enunciado de la práctica para la convocatoria extraordinaria. Como en el resto de prácticas de la asignatura, para que la nota se guarde hasta el examen extraordinario de Julio se debe sacar una nota mayor o igual que cinco.

Grupos: Los grupos estarán formados por dos alumnos matriculados en el mismo semestre. Los alumnos deben estar matriculados y dados de alta en el sistema de entrega (maui) antes de proceder a realizar el registro del grupo. Ninguno de los dos alumnos deberá haber realizado entrega alguna de la práctica antes de definir el grupo. Una vez creado el grupo es el mismo para todas las prácticas en grupo del semestre.

El alumno debe estar registrado en el sistema de entrega. El registro se hace una sola vez por semestre.

http://maui.ls.fi.upm.es/entrega/formreg.php

El grupo se crea por medio de la URL:

http://maui.ls.fi.upm.es/entrega/CrearGruposProgramacion2.html

El alumno de un grupo que realice la primera entrega de una práctica, será el que tenga que hacer todas las entregas para esa práctica.

Entrega: La práctica se entregará a través de la página web:

http://maui.ls.fi.upm.es/entrega/

La práctica entregada debe compilar en la versión 1.6 del J2SE de Oracle/Sun y debe ser compatible con JUnit 4.8. En el momento de realizar la entrega, la práctica será sometida a una serie de pruebas que deberá superar para que la entrega sea admitida. El

alumno dispondrá de **un número máximo de diez entregas**. Ahora bien, si el alumno realiza **más de cinco** entregas, se le **restará un punto** en la nota final de la práctica. Asimismo, por el hecho de que la práctica sea admitida, eso no implicará que la práctica esté aprobada. **El fichero a entregar será practica.jar con la estructura dada y sólo con el código fuente y la documentación generada por javadoc**.

Fecha límite: Es el día **11-5-2012** a las **9:00 AM**.

Código Auxiliar: La realización de esta práctica requiere la utilización de los ficheros auxiliares que se pueden encontrar dentro del fichero **cod_alumnosTaller.zip** que acompaña a este enunciado. Las implementaciones de los TADs que se pueden utilizar se encuentran en el fichero TADS.jar que podrán encontrar en moodle en materiales de la asignatura/lib/TADs.

Autoevaluación: El alumno debe comprobar que su ejercicio no contiene ninguno de los errores explicados en el apartado 5 de este enunciado. **Si el ejercicio contiene alguno de estos errores, se calificará como suspenso**.

Detección Automática de Copias: Cada práctica entregada se comparará con el resto de prácticas entregadas en todos los grupos de la asignatura. Esto se realizará utilizando un sofisticado programa de detección de copias.

Consecuencias de haber copiado: Todos los alumnos involucrados en una copia, bien por copiar o por ser copiados, quedan inhabilitados para presentarse a todas las convocatorias de examen del presente curso, además de la posible apertura de expediente académico.

En esta práctica se pide implementar una simulación del funcionamiento de un taller de reparaciones de coches. Más concretamente, se pretende simular la asignación de reparaciones a los mecánicos y el trabajo realizado por los mecánicos al reparar los coches.

Dentro de esta simulación destacan los siguientes tipos de objetos: el **taller** propiamente dicho, los **mecánicos** que trabajan en él, y las **reparaciones** que se realizan. El taller es el principal objeto de esta simulación, y es el encargado de asignar reparaciones a sus mecánicos.

A grandes rasgos, el funcionamiento del taller que se pretende simular será el siguiente. Cuando llegue al taller un coche que requiere una reparación, el (objeto) taller debe intentar asignar dicha reparación a un (objeto) mecánico. Si, por los motivos que veremos a continuación, no existe ningún mecánico disponible para hacerse cargo de esa reparación, la reparación quedará en una cola de espera hasta que un mecánico pueda hacerse cargo de ella. Si, por el contrario, existe un mecánico disponible (bajo las condiciones que se explicarán más adelante), el taller asignará la reparación a dicho mecánico.

A continuación, se va a explicar de manera más detallada en qué orden se van a realizar las reparaciones que vayan llegando al taller.

1. Asignación y Realización de Reparaciones

El taller distingue entre dos tipos de reparaciones, **normales** y **con prioridad**:

- **Reparación normal** va a ser un tipo de reparación cuya realización puede llevar muchas horas.
- **Reparación con prioridad** es un tipo de reparación que, al contrario que una reparación normal, se puede realizar en poco tiempo. Una reparación con prioridad tendrá asociada una prioridad que representaremos mediante un valor numérico comprendido entre 1 y MAXPRIORIDAD, en donde el **valor 1 representará la prioridad mínima**, y el **valor MAXPRIORIDAD la máxima**. Esta prioridad será tenida en cuenta por los mecánicos y el taller a la hora de decidir si una reparación es atendida o debe esperar. El tiempo requerido (ciclos de trabajo) para una reparación prioritaria lo determina el taller, para lo que utiliza la siguiente fórmula (MAXPRIORIDAD+1) – P, donde P es la prioridad de la reparación.

Una reparación en el taller se puede encontrar en los siguientes estados:

- **En reparación**: La tiene un mecánico asignada, y no se encuentra en ninguna de las zonas de espera del taller.

- **En espera**: Es una reparación que ha llegado al taller, pero que aún no ha sido iniciada. Se encuentra en la zona de espera establecida para este tipo de reparaciones.

- **Pospuesta**: Es una reparación que fue asignada a un mecánico, pero que el mecánico tuvo que posponer, porque tuvo que atender otra reparación más urgente. La reparación espera a ser retomada por cualquier mecánico disponible en una zona especial para las reparaciones pospuestas.

- **Reparación completada**: Reparación que ha sido terminada.

La asignación y realización de una reparación que llega al taller se tratará de la siguiente manera atendiendo al tipo de la reparación:

- **Si la reparación es normal**, el taller comprobará primero si ya hay otras reparaciones **pospuestas** o **en espera**, y si es así, guardará la reparación en la zona de espera de reparaciones normales. En caso contrario, buscará un mecánico que esté desocupado y le asignará dicho mecánico. Si hay más de un mecánico desocupado, se le asignará al que tenga menor identificador. En cambio, si no hay ningún mecánico desocupado, la reparación se guardará en la zona de espera de reparaciones normales en el objeto taller.

- **Si la reparación es con prioridad**[1], el taller actuará de la siguiente manera:

    1. Si hay reparaciones **pospuestas** o **en espera**, y alguna tiene **mayor o igual prioridad** que la nueva reparación, se guarda la nueva reparación en la zona de espera correspondiente.

    2. Si todos los mecánicos están ocupados con reparaciones de mayor prioridad que la nueva, se guarda también la nueva reparación en la zona de espera correspondiente.

    3. En caso contrario, se procederá a seleccionar el mecánico más idóneo para realizar esta reparación de acuerdo a este orden de preferencia:

        3.1. El mecánico desocupado con menor identificador,

        3.2. El mecánico ocupado que pueda encargarse de dicha reparación (según el método **puedoHacerReparacion**) y que esté realizando la reparación con menor prioridad de todas las asignadas. En caso de haber varios mecánicos que reúnan estos requisitos, se selecciona el de menor identificador.

Se debe entender como una descripción de lo que se debe hacer. El alumno debe determinar el orden más adecuado para que se cumpla lo especificado

Si un mecánico estaba ocupado y recibe la asignación de una nueva reparación (siempre más urgente que la que estaba realizando)**, pospondrá la reparación que esté realizando y se la pasará al taller** para que la ubique en la zona de reparaciones pospuestas, y se pondrá a trabajar inmediatamente en la reparación que se le acaba de asignar.

2. Diseño de la aplicación

Para comprobar el correcto funcionamiento del taller se proporciona el JUnit TestTaller.java, que contiene las pruebas que debe superar la implementación.

La jornada laboral en el taller estará dividida en un cierto número de ciclos de trabajo. En un ciclo de trabajo cada mecánico del taller que se encuentre ocupado con una reparación, progresará en la realización de dicha reparación. Cada reparación requerirá un cierto número de ciclos de trabajo para ser completada.

El taller actúa como supervisor de los trabajos que realizan sus mecánicos. Los mecánicos informan al supervisor de sus progresos y de los hechos relevantes.

El taller no 'recuerda' las reparaciones que están en progreso (**estado en reparación**), ya que es responsabilidad del mecánico que la está atendiendo.

SupervisorTrabajo

Interfaz que establece los servicios que se utilizarán para supervisar el trabajo de los mecánicos. Declara los servicios:

- **notificarReparacionFinalizada**(mecanico : Mecanico, reparacion : Reparacion) : void: Que será invocado por un mecánico cuando concluye una reparación, para avisar a su supervisor de que ha terminado con la reparación pasada como parámetro. También se encargará de asignar una nueva reparación al mecánico si hubiera reparaciones **pospuestas o en espera**. Para ello el taller considerará la **última** reparación pospuesta y la reparación en espera de mayor prioridad que lleve más tiempo esperando, y elegirá de entre esas dos la reparación de mayor prioridad. En caso de empate se elegirá la pospuesta.

- **notificarReparacionPospuesta**(mecanico: Mecanico, reparacion : Reparacion) : void: Método que invoca el mecánico para indicar al taller que se pospone la reparación pasada como argumento. El taller debe recoger dicha reparación y ubicarla donde corresponda.

Taller

Clase que implementa la gestión de las reparaciones y los mecánicos asignados al taller. Además actúa como supervisor de los mecánicos, es decir, también implementa los

métodos definidos por la interfaz **SupervisorTrabajo**. Proporciona una serie de servicios. Estos son:

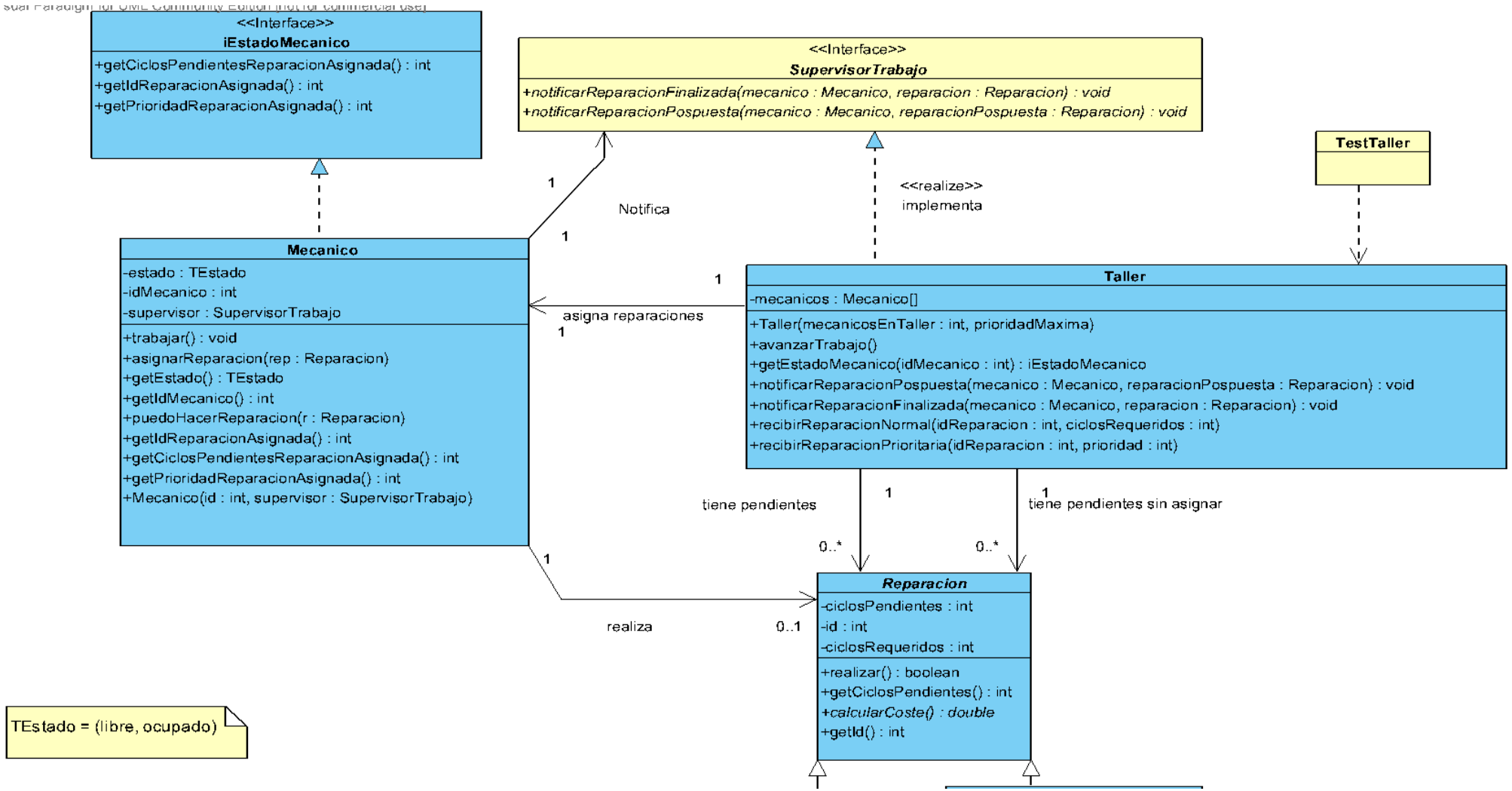
- **Taller**: es el constructor que recibe como parámetros el número de mecánicos que hay en el taller y la prioridad máxima de una reparación que puede admitir. Si el número de mecánicos es menor que uno, se producirá ErrorNumeroDeMecanicos, y si el número de prioridades es menor que cero, se producirá ErrorNumeroDePrioridades.

- **avanzarTrabajo**: Método que se invoca para indicar que empieza un nuevo ciclo de trabajo. Debe indicar a cada mecánico ocupado que realice un ciclo de trabajo.

- **recibirReparacionPrioritaria**: Método que se encarga de recibir una reparación con prioridad y asignársela a un mecánico, o ponerla en espera si no hay mecánicos disponibles. La asignación de un mecánico se explica en el apartado “”. Si la prioridad dada no es un valor válido se producirá ErrorPrioridadIncorrecta.

- **recibirReparacionNormal**: Método que se encarga de recibir una reparación normal y asignársela a un mecánico, o ponerla en espera si no hay mecánicos desocupados. La asignación de un mecánico se explica en el apartado “”. Si el número de ciclos suministrado no es correcto se producirá ErrorNumeroCiclosIncorrecto.

- **getEstadoMecanico**: Método que recibe el identificador de un mecánico y retorna una instancia que implementa la interfaz iEstadoMecanico y que proporciona la información relevante del mecánico indicado (estado, reparación asignada y ciclos pendientes). Si el identificador del mecánico no hace referencia a un identificador válido se levanta la excepción ErrorIdentificadorMecanicoNoValido.


Visual Paradigm for UML Community Edition [not for commercial use]
<<Interface>>
iEstadoMecanico
+getCiclosPendientesReparacionAsignada() : int
+getIdReparacionAsignada() : int
+getPrioridadReparacionAsignada() : int
<<Interface>>
SupervisorTrabajo
+notificarReparacionFinalizada(mecanico : Mecanico, reparacion : Reparacion) : void
+notificarReparacionPospuesta(mecanico : Mecanico, reparacionPospuesta : Reparacion) : void
TestTaller
<<realize>>
implementa
Notifica
1
1
Mecanico
-estado : TEstado
-idMecanico : int
-supervisor : SupervisorTrabajo
+trabajar() : void
+asignarReparacion(rep : Reparacion)
+getEstado() : TEstado
+getIdMecanico() : int
+puedoHacerReparacion(r : Reparacion)
+getIdReparacionAsignada() : int
+getCiclosPendientesReparacionAsignada() : int
+getPrioridadReparacionAsignada() : int
+Mecanico(id : int, supervisor : SupervisorTrabajo)
1
asigna reparaciones
1
Taller
-mecanicos : Mecanico[]
+Taller(mecanicosEnTaller : int, prioridadMaxima)
+avanzarTrabajo()
+getEstadoMecanico(idMecanico : int) : iEstadoMecanico
+notificarReparacionPospuesta(mecanico : Mecanico, reparacionPospuesta : Reparacion) : void
+notificarReparacionFinalizada(mecanico : Mecanico, reparacion : Reparacion) : void
+recibirReparacionNormal(idReparacion : int, ciclosRequeridos : int)
+recibirReparacionPrioritaria(idReparacion : int, prioridad : int)
tiene pendientes
1
0..*
1
tiene pendientes sin asignar
0..*
1
realiza
0..1
Reparacion
-ciclosPendientes : int
-id : int
-ciclosRequeridos : int
+realizar() : boolean
+getCiclosPendientes() : int
+calcularCoste() : double
+getId() : int
TEstado = (libre, ocupado)

3. Código de apoyo

Los alumnos deben utilizar el código de apoyo que se encuentra en el archivo cod_alumnosTaller.zip. Este archivo contiene:

- mecanico.jar: Librería que tiene la implementación de la clase mecánico y todas las excepciones asociadas
- reparacion.jar: Librería que tiene la implementación de la clase reparación y todas las excepciones asociadas
- src/taller: Directorio en el que se encuentra el código del paquete taller, y donde se debe ubicar Taller.java
- src/taller/excepciones: Contiene las implementaciones de las excepciones necesarias
- src/principal: TestTaller.java. JUnit que sirve para probar la implementación del taller. Este JUnit requiere JUnit 4.8

4. Errores graves a evitar

En esta sección se enumeran algunos errores que el alumno debe evitar. Esta lista no es exhaustiva en el sentido de que no recoge todos los posibles errores que el alumno debe evitar cometer.

1. Se declaran atributos públicos.
2. Utiliza atributos de clase o static (para esta práctica no son necesarios).
3. Se realizan operaciones de entrada/salida en alguna de las clases implementadas por el alumno, excepto en las ramas catch en donde sí que se permite.

5. Consideraciones a la hora de entregar la práctica

Se deben comentar las clases y los métodos utilizando comentarios reconocidos por javadoc. Los comentarios javadoc de los métodos deben tener la misma estructura que los utilizados en las implementaciones de los TADs vistos en clase.

Se debe entregar el fichero jar (**practica.jar**) con la estructura que se muestra en la Figura 1 y respetando los paquetes indicados en el código de apoyo. No debe incluir ficheros .class, ni los jar suministrados en el código de apoyo. Se deben entregar los html generados por medio de javadoc para el paquete taller.

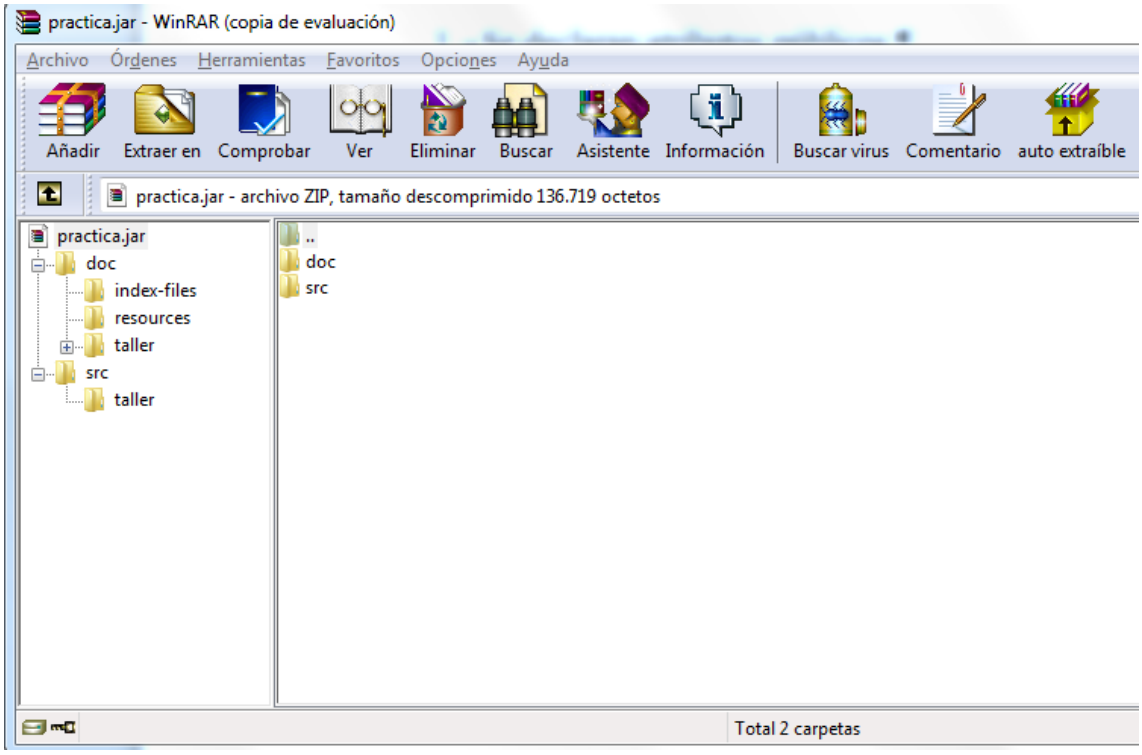


Figura 1 Estructura de practica.jar

Sólo se pueden usar los TADs que hay disponibles en TADS.jar que se encuentra disponible en moodle.

Se valora positivamente:

1. La utilización de nombres significativos para los identificadores, así como la utilización de los convenios de Java.
2. La utilización de métodos auxiliares que implementen tareas comunes a varios métodos, y que de esa forma eviten la duplicidad de código.
3. La correcta indentación del código. Se recomienda la utilización en eclipse del atajo de teclado CTRL + i.
4. La implementación de un código eficiente que no realice operaciones innecesarias.