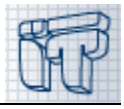


NOMBRE:
APELLIDOS:
NIA:
GRUPO:

2ª Parte: Problemas (7 puntos sobre 10)

Duración: 180 minutos
Puntuación máxima: 7 puntos
Fecha: 27 junio 2019

Instrucciones para el examen:

- No se permite el uso de libros o apuntes, ni tener teléfonos móviles u otros dispositivos electrónicos encendidos. Incumplir cualquiera de estas normas puede ser motivo de expulsión inmediata del examen.
- Rellena tus datos personales antes de comenzar a realizar el examen.
- El examen debe rellenarse con bolígrafo azul o negro. No está permitido entregar el examen con lapicero.

Problema 1 (2,25 puntos)

Una empresa que comercializa un servicio de correo electrónico necesita implementar un sistema de autenticación de acceso a su servicio. El servicio de correo electrónico ofrece dos tipos de cuentas: una cuenta de propósito general que puede utilizarse para uso personal modelada en la clase `Account` y otra profesional modelada en la clase `ProfessionalAccount`.

Apartado 1.1 (0,75 puntos)

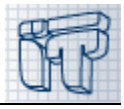
Se requiere implementar la clase que modela la cuenta de propósito general (`Account`), de forma que cumpla los siguientes requisitos:

- La clase contendrá el nombre de usuario (`username`) y la contraseña (`password`). Ambos atributos podrán contener caracteres alfanuméricos. Asimismo, la clase contendrá un atributo para determinar si la cuenta está bloqueada o no (`isBlocked`). Los atributos no serán accesibles directamente desde ninguna otra clase.
- La contraseña debe tener una longitud mínima de 8 caracteres. Esta longitud mínima debe modelarse como una constante (`PASSWORD_MIN_LENGTH`). Si se trata de establecer una contraseña con una longitud menor de 8 caracteres debe lanzarse la excepción `PasswordException` con el mensaje *"The length of the password must be at least 8 characters"*.
- La clase `PasswordException` ya se encuentra implementada y se puede usar sin necesidad de implementarla.
- El constructor recibe como parámetros el nombre de usuario y la contraseña. La contraseña debe cumplir con el requisito de longitud mínima previamente mencionado.
- Se requieren los métodos de acceso necesarios (`get` y `set`) teniendo en cuenta que se puede acceder (`get`) a todos los atributos, pero sólo se pueden modificar (`set`) el atributo que indica si la cuenta está bloqueada o no y la contraseña (la cual siempre debe cumplir con la longitud mínima previamente mencionada).

Apartado 1.2 (0,5 puntos)

Se requiere implementar la clase que modela la cuenta profesional (`ProfessionalAccount`) teniendo en cuenta que se trata de un tipo específico de cuenta (`Account`) con las siguientes particularidades:

- Además de los atributos de `Account`, esta clase almacena el nombre de la empresa (`company`) a la que pertenece el usuario. El nombre de la empresa será inicializado en el constructor y no se podrá cambiar tras la creación de la cuenta de usuario, aunque su valor sí podrá ser recuperado desde otras clases.



- La contraseña tiene la misma restricción de longitud mínima que en el caso de la clase `Account`, aunque incluye una validación de seguridad adicional por la que el valor de la contraseña no puede ser igual al valor del `username`.

Apartado 1.3 (1 punto)

Se requiere implementar la clase `AuthenticatorManager` que mantiene una lista de cuentas de usuario de correo electrónico con independencia de si son cuentas personales o profesionales y se encarga de realizar la autenticación del usuario (comprobación de que el nombre de usuario y contraseña introducidos coinciden con los que se encuentran almacenados). La clase `AuthenticatorManager` implementa la siguiente interfaz:

```
public interface Authenticator {

    static final int USER_AUTHENTICATED = 0;
    static final int PASSWORD_INCORRECT = 1;
    static final int ACCOUNT_BLOCKED = 2;
    static final int ACCOUNT_NOT_FOUND = 3;

    int authenticateUser(String username, String password);
}
```

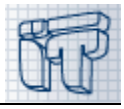
La clase `AuthenticatorManager` debe cumplir los siguientes requisitos:

- Recibe en el constructor un array con todas las cuentas registradas en el servicio (`Account[] accounts`).
- El método `authenticateUser(...)` se encarga de la autenticación del usuario que recibe por parámetro, utilizando para ello el array de cuentas del servicio recibido en el constructor. Devuelve un entero indicando una de las 4 posibilidades del proceso de autenticación establecidas en la interfaz `Authenticator`. La seguridad es un aspecto muy importante para la empresa, así que cuando el usuario introduce una contraseña incorrecta una vez, la cuenta se debe bloquear automáticamente.
- Para la implementación del método `authenticateUser(...)` se debe tener en cuenta que no habrá dos cuentas con el mismo nombre de usuario.

Problema 2 (0,75 puntos)

Se proporciona a continuación la clase `Number`:

```
public class Number {
	private int number;
	public Number (int number){
		this.number = number;
	}
	public int result(){
		if (number > 0) {
			return 1;
		} else if (number < 0) {
			return -1;
		} else {
			return 0;
		}
	}
}
```



Se pide implementar las siguientes pruebas de programa (tests). En caso de que alguna prueba no sea posible de implementar, justifique el porqué. **Nota:** Sólo se puede crear un máximo de un objeto por apartado y no se pueden realizar varias llamadas al mismo método en el mismo apartado.

Apartado 2.1 (0,45 puntos)

Programe un test que alcance una **cobertura de ramas** entre 1% (incluido) y 33% (sin incluir).

Apartado 2.2 (0,15 puntos)

Programe un test que alcance una **cobertura de ramas** entre 50% (sin incluir) y 67% (incluido).

Apartado 2.3 (0,15 puntos)

Programe un test que alcance una **cobertura de métodos** de entre el 1 y el 50% (ambos incluidos).

Problema 3 (2 puntos)

La empresa *"Paquetes Programados"* se dedica a la entrega de paquetes a particulares. Para organizar y gestionar la recepción y el envío de los paquetes, la empresa ha encargado a su equipo de programadores la creación de un software que facilite y automatice sus tareas.

La tarea del equipo programador es comenzar a implementar una cola de paquetes para gestionar su recepción y posterior envío. El equipo proporciona las clases ya programadas que puedes observar a continuación y te encomienda la tarea de empezar a crear una nueva clase `PacketsQueue` que herede de `LinkedQueue<Packet>`. Esta nueva clase `PacketsQueue` será la implementación que realice la empresa de su propia cola de paquetes.

```
public class Node<E>{
   private E info;
   private Node<E> next;
   public Node(){…}
   public Node(E info){…}
   public Node(E info, Node<E> next){…}
   public E getInfo(){…}
   public void setInfo(E info){…}
   public Node<E> getNext(){…}
   public void setNext(Node<E> next){…}
}
```

```
public interface Queue<E>{
   boolean isEmpty();
   int size();
   E front();
   void enqueue (E info);
   E dequeue();
}
```

```
public class Packet{
   private int numberId;
   private String dest;
   private boolean urgent;
   public Packet(int numberId, String dest, boolean urgent){
      this.numberId = numberId;
      this.dest = dest;
      this.urgent = urgent;
   }
   public int getNumberId(){ return numberId; }
   public void setNumberId(int numberId){
      this.numberId = numberId;
   }
   public String getDest(){ return dest; }
   }
   public void setDest(String dest){
      this.dest = dest;
   }
   public boolean isUrgent(){ return urgent; }
   public void setUrgent(boolean urgent){
      this.urgent = urgent;
   }
}
```

```
public class LinkedQueue<E> implements
Queue<E>{
   protected Node<E> top;
   protected Node<E> tail;
   protected int size;

   public LinkedQueue(){…}

   public boolean isEmpty(){…}
   public int size(){…
   public E front(){…}
   public void enqueue (E info){…}
   public E dequeue(){…}
}
```

```
public class PacketsQueue extends LinkedQueue<Packet> {
  public PacketsQueue(){
        super();
  }
  public Packet sendPacket(){ //APARTADO 3.1 }
  public Packet sendUrgent(){ //APARTADO 3.2 }
}
```

Apartado 3.1 (0,4 puntos)

Programe el método `public Packet sendPacket()`. Este método devuelve el primer paquete que esté en espera para ser enviado y lo elimina de la cola. Si no hay ningún paquete pendiente (la cola está vacía) el valor devuelto debe ser `null`.

Apartado 3.2 (1,6 puntos)

Programe el método `public Packet sendUrgent()`. Este método busca el primer paquete de la cola que sea urgente, lo devuelve y lo elimina de la cola. Si en la cola no existe ningún paquete urgente, el valor devuelto debe ser `null`.

Problema 4 (2 puntos)

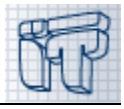
Para árboles binarios, se define el **factor de equilibrio** (*balance factor*) de cada nodo como la diferencia entre la altura de su subárbol derecho y la altura de su subárbol izquierdo, de forma que:

- Un factor de equilibrio 0 indica que los dos subárboles tienen la misma altura.
- Si el factor de equilibrio es positivo indica que el subárbol derecho tiene mayor altura que el subárbol izquierdo.
- Si el factor de equilibrio es negativo indica que el subárbol izquierdo tiene mayor altura que el subárbol derecho.

Para ello, se ha añadido un atributo `private int balanceFactor` con sus correspondientes *getter* y *setter* públicos (`getBalanceFactor` y `setBalanceFactor` respectivamente) a la clase `LBNode<E>`, estando dichos métodos ya implementados. Inicialmente todos los nodos partirán de un factor de equilibrio con valor 0, siendo necesario llamar a un método de actualización de todos los factores de equilibrio cuando se termine de construir un árbol dado.

Se pide añadir a `LBTree<E>` un método `public void updateBalanceFactor()` que actualice recursivamente el factor de equilibrio en todos los nodos del árbol.

Se recuerdan los métodos disponibles en la interfaz `BTree<E>` y en las clases `LBNode<E>` y `LBTree<E>`:



```java
public interface BTree<E> {

  static final int LEFT = 0;
  static final int RIGHT = 1;

  boolean isEmpty();
  E getInfo();
  BTree<E> getLeft();
  BTree<E> getRight();
  void insert(BTree<E> tree, int side);
  BTree<E> extract(int side);
  String toStringPreOrder();
  String toStringInOrder();
  String toStringPostOrder();
  String toString();

  int size();
  int height();
  boolean equals(BTree<E> tree);
  boolean find(BTree<E> tree);
}
```

```java
public class LBNode<E>{

  private E info;
  private BTree<E> left;
  private BTree<E> right;
  private int balanceFactor;

  LBNode(E info, BTree<E> left, BTree<E> right) {
    this.left = left;
    this.right = right;
    this.info = info;
  }

  E getInfo() { return info; }
  void setInfo(E info) { this.info = info; }

  BTree<E> getLeft() { return left; }
  void setLeft(BTree<E> left) { this.left = left; }

  BTree<E> getRight() { return right; }
  void setRight(BTree<E> right) { this.right = right; }

  int getBalanceFactor() { return balanceFactor; }
  void setBalanceFactor(int balanceFactor){
    this.balanceFactor = balanceFactor;
  }
}
```

```java
public class LBTree<E> implements BTree<E>{
   private LBNode<E> root;
   public LBTree() {
      root = null;
   }
   public LBTree(E info) {
      root = new LBNode<E>(info, new LBTree<E>(), new LBTree<E>());
   }
   ...
   public void updateBalanceFactor() { //Problema 4 }
}
```

SOLUCIONES DE REFERENCIA (Varias soluciones a cada uno de los problemas son posibles)

PROBLEMA 1

Apartado 1.1 (0,75 puntos)

```
public class Account {
    private String username;
    private String password;
    private boolean isBlocked;

    protected static final short PASSWORD_MIN_LENGTH = 8;
    public Account(String username, String password) throws PasswordException {
        this.username = username;
        setPassword(password);
        isBlocked = false;
    }
    public String getUsername() {
        return username;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) throws PasswordException {
        if (password.length() < PASSWORD_MIN_LENGTH) {
            throw new PasswordException("The length of the password must be at least
                8 characters");
        }
        this.password = password;
    }
    public boolean getIsBlocked() {
        return isBlocked;
    }
    public void setIsBlocked(boolean isBlocked) {
        this.isBlocked = isBlocked;
    }
}
```

Apartado 1.2 (0,5 puntos)

```
public class ProfessionalAccount extends Account {
    private String company;
    public ProfessionalAccount(String username, String password, String company)
            throws PasswordException {
        super(username, password);
        this.company = company;
    }

    public String getCompany() {
        return company;
    }
    @Override
    public void setPassword(String password) throws PasswordException {
        if (password.equals(getUsername())) {
            throw new PasswordException("The password cannot contain the username");
        }
```

```
        super.setPassword(password);
    }
}
```

Apartado 1.3 (1 punto)

```
public class AuthenticationManager implements Authenticator {
    private Account[] accounts;

    public AuthenticationManager(Account[] accounts) {
        this.accounts = accounts;
    }

    public int authenticateUser(String username, String password) {
        Account account = null;
        boolean found = false;
        for (int i = 0; i < accounts.length && !found; i++) {
            if (accounts[i].getUsername().equals(username)) {
                account = accounts[i];
                found = true;
            }
        }
        int authenticationResult = PASSWORD_INCORRECT;
        if (account == null) {
            authenticationResult = ACCOUNT_NOT_FOUND;
        } else if (account.getIsBlocked()) {
            authenticationResult = ACCOUNT_BLOCKED;
        } else if (account.getPassword().equals(password)) {
            authenticationResult = USER_AUTHENTICATED;
        } else {
            account.setIsBlocked(true);
        }
        return authenticationResult;
    }
}
```

Apartado 1.1 (0,75 puntos)

- 0,05: Declaración de la clase.
- 0,1: Atributos correctos (tipo y visibilidad).
- 0,05: Constante correcta (tipo y final). No se tiene en cuenta la visibilidad asignada.
- 0,2: Constructor.
  - 0,05: Definición del método con la excepción.
  - 0,05: Inicialización de username y de isBlocked a false (si no asignan false explícitamente también es correcto dado que isBlocked por defecto es false).
  - 0,1: Inicialización de la contraseña con la validación, ya sea invocando a un método set o realizando la comprobación del número de caracteres dentro del constructor.
- 0,1: Getters y setters (excepto setPassword).
- 0,25: setPassword.
  - 0,05: Definición del método con la excepción que puede ser lanzada.
  - 0,1: Comprobación de la longitud.
  - 0,05: Lanzamiento de la excepción.
  - 0,05: Asignación del atributo.

Apartado 1.2 (0,5 puntos)

- 0,1: Declaración de la clase.
  - -0,05 si no hereda de Account.
- 0,05: Atributo company y getter correspondiente
- 0,15: Constructor.
  - 0,05: Definición del constructor. Si no define la excepción, pero ya se ha penalizado en Account no se debe penalizar nuevamente.
  - 0,05: Invocación constructor padre (super).
  - 0,05: Inicialización de company.
- 0,2: setPassword.
  - 0,1: Comprobación de username y lanzamiento de la excepción
  - 0,1: Invocación al método set del padre
  - Si se invoca al set del padre antes de hacer la validación de que no contiene el username -0,1.

Apartado 1.3 (1 punto)

- 0,1: Declaración de la clase.
- 0,1: Atributo accounts.
- 0,15: Constructor.
- 0,65: Método authenticateUser.
  - 0,35: Búsqueda de la cuenta en el array.
    - -0,05 si usa == en lugar de equals.
    - -0.15 si no se para el bucle al encontrar la cuenta
  - 0,05: Condición de usuario no encontrado.
  - 0,05: Condición de cuenta bloqueada.
  - 0,1: Condición de usuario autenticado correctamente
  - 0,1: Condición de contraseña incorrecta y bloqueo de la cuenta.
    - -0,05 si no bloquea la cuenta, pero sí devuelve el valor correcto o viceversa
  - Si no devuelve nada -0.1.

PROBLEMA 2 (0,75 puntos)

Apartado 2.1 (0,45 puntos)

```
@Test
public void testSectionA(){
    Number number = new Number(5);
    assertEquals(number.result(),1);
}
```

Apartado 2.2 (0,15 puntos)

```
//No se puede hacer
```

No se puede hacer porque:

1. Al haber 4 ramas, la cobertura solo puede ser 0%, 25%, 50%, 75% o 100%, dependiendo del número de ramas que se cubran.
2. Se pide una cobertura de entre 50% (sin incluir) y 67% (incluido). Por tanto, no hay ningún valor posible en ese rango.

Las 4 ramas son: las ramas verdadera y falsa del if (si el número > 0 o no), y las ramas verdadera y falsa del else-if (si el número < 0 o no).

Apartado 2.3 (0,15 puntos)

```
@Test
public void testSectionC(){
    Number number = new Number(5);
}
```

Apartado 2.1 (0,45 puntos)

- 0,05: Correcta cabecera del método test (con cualquier nombre elegido para el método)
- 0,1: Correcta creación del objeto de la clase Number: `Number number = new Number(5)`
- 0,3: Correcto assertEquals.
- Si el código está bien pero el test no alcanza la cobertura deseada puntuar con 0,2 todo el apartado.

Apartado 2.2 (0,15 puntos)

- 0,15: Si se justifica por qué no se puede hacer. Una posible justificación se incluye en la solución proporcionada más arriba.
- Si sólo pone "No se puede hacer" sin justificar, puntuar con 0,05 todo el apartado.

Apartado 2.3 (0,15 puntos)

- 0,05: Correcta cabecera del método test (con cualquier nombre elegido para el método):
- 0,1: Correcta creación del objeto de la clase Number: `Number number = new Number(5).`
- Si el código está bien pero el test no alcanza la cobertura deseada puntuar con 0,05 todo el apartado.

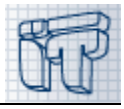
PROBLEMA 3

Apartado 3.1 (0,4 puntos)

```
public Packet sendPacket(){
    return this.dequeue();
}
```

Apartado 3.2 (1,6 puntos)

```
public Packet sendUrgent(){
    if(top!=null){
        Node<Packet> tmp = top.getNext();
        Node<Packet> prev = top;
        if(prev.getInfo().isUrgent()){
            return this.dequeue();
        }else{
            while(tmp!=null){
                if(tmp.getInfo().isUrgent()){
                    prev.setNext(tmp.getNext());
                    size--;
                    if(tmp.getNext()==null){
                        tail = prev;
                    }
                    return tmp.getInfo();
                }else{
                    prev = tmp;
                    tmp = tmp.getNext();
```



```
                    }
                }
            }
        }
        return null;
}
```

Apartado 3.1 (0,4 puntos)

- 0,4: Manejo correcto de la operación dequeue, de forma manual o mediante el método dequeue():

  De forma manual:

  - 0,1: Actualizar top.
  - 0,1: Actualizar tail.
  - 0,1: Actualizar size.
  - 0,1: Devolver el paquete.

  Mediante método dequeue():

  - 0,3: Invocar al método dequeue().
  - 0,1: Devolver el paquete.

Apartado 3.2 (1,6 puntos)

- 0,4: Devuelve null si no hay paquetes urgentes
- 0,4: Maneja correctamente el caso en el que el primer urgente sea top, de forma manual o mediante el método dequeue():

  De forma manual:

  - 0,1: Actualizar top
  - 0,1: Actualizar tail
  - 0,1: Actualizar size
  - 0,1: Devolver el paquete

  Mediante método dequeue():

  - 0,3: Invocar al método dequeue()
  - 0,1: Devolver el paquete

- 0,4: Maneja correctamente el caso en el que el primer urgente está en medio de la cola.
  - 0,2: Actualizar referencias next
  - 0,1: Actualizar size
  - 0,1: Devolver el paquete

- 0,4: Maneja correctamente el caso en el que el primer urgente sea tail.
  - 0,1: Actualizar referencias next
  - 0,1: Actualizar tail
  - 0,1: Actualizar size
  - 0,1: Devolver el paquete

PROBLEMA 4 (2 puntos)

```
public void updateBalanceFactor() {
  if (!isEmpty()) {
    this.root.setBalanceFactor(this.getRight().height() - this.getLeft().height());
    ((LBTree<E>) this.getLeft()).updateBalanceFactor();
    ((LBTree<E>) this.getRight()).updateBalanceFactor();
  }
}
```

- 0,25: Determinar la condición del caso base.
- 0,25: Resolver el caso base (es decir: por no hacer ninguna modificación a un árbol vacío).
- 0,50: Calcular el factor de equilibrio del propio nodo correctamente (es decir: por calcular la diferencia entre la altura del subárbol derecho - altura del subárbol izquierdo). Si se hace en sentido inverso (altura izq - altura dcha) entonces valorar con un máximo de 0,40.
- 0,50: Almacenar dicho factor usando el método setBalanceFactor sobre el objeto this.root. Si se usa el atributo privado balanceFactor y no el setter entonces valorar con un máximo de 0,40.
- 0,25: Llamar recursivamente al método para el subárbol izquierdo.
- 0,25: Llamar recursivamente al método para el subárbol derecho.
- Penalizar con -0,05 si no realiza casting alguno a LBTree<E> para poder llamar al método recursivo (tanto si falta uno como si faltan ambos castings, penalizar únicamente con -0,05).