Práctica #1

Patrón de diseño Singleton

1.1 Objetivos

El objetivo de esta primera práctica es consolidar los conceptos vistos en teoría sobre el PD *Singleton*. Para ello es aconsejable que haya asistido a las clases previas de teoría y haya repasado los apuntes de las mismas.

Vamos a utilizar este patrón con dos objetivos distintos:

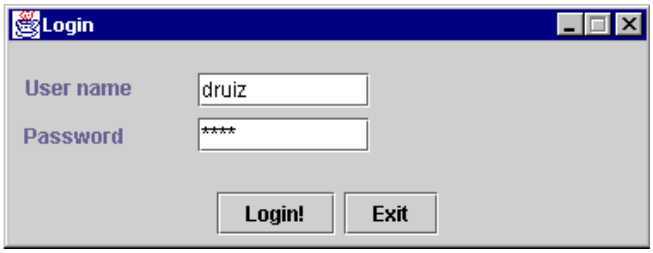
i. Para almacenar información sobre el estado de la aplicación.

ii. Para asegurar que a un determinado recurso sólo tiene acceso un objeto y además en exclusión mutua.

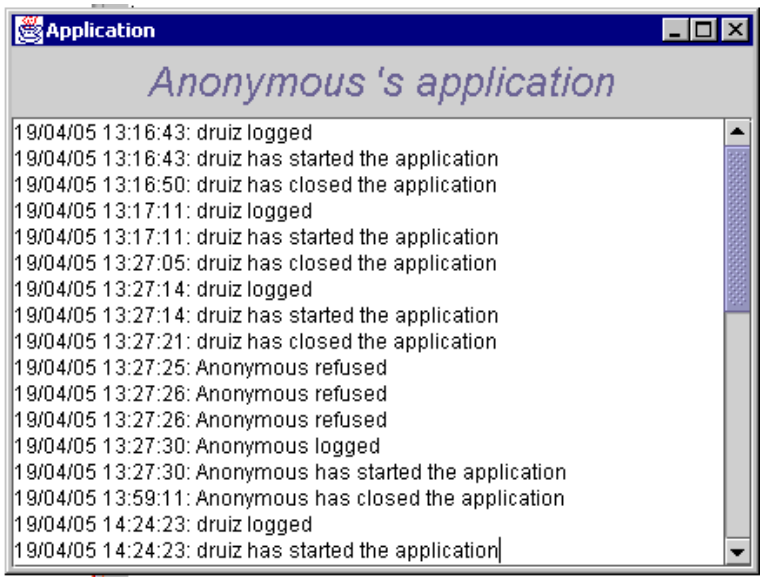
1.2 Práctica a realizar

Se va a realizar una aplicación que primero solicite al usuario su nombre y una contraseña (vea la Figura 1.1.(a)). Si la autenticación es correcta mostrará una ventana en la que aparecerá información sobre quién y cuándo ha ejecutado la aplicación (vea la Figura 1.1.(b)). Si la autenticación falla de momento no haremos nada, será en una práctica posterior donde implementaremos distintas políticas de rechazo.

El siguiente caso de uso recoge cómo los usuarios se autentican para usar la aplicación:



(a) Ventana de autenticación



(b) Ventana de aplicación

Figura 1.1: *Interfaz de usuario.*

UC–001	**Autenticación de usuarios**	
Versión	*1.0 (Abril/2005)*	
Autores	• *David Ruiz (Universidad de Sevilla)*	
Descripción	El sistema deberá comportarse tal como se describe en el siguiente caso de uso cuando *un usuario se autentique.*	
Precondición	*Ninguna*	
Secuencia normal	**Paso**	**Acción**
	1	El sistema *solicita al usuario que introduzca su nombre de usuario y su clave.*
	2	El actor *usuario introduce su nombre de usuario y contraseña y pulsa el botón "Login!".*
	3	Si *la autenticación es correcta*, el sistema *permitirá el acceso a la aplicación principal*
	4	Si *la autenticación no es correcta*, el sistema *no permitirá el acceso a aplicación*
	5	El sistema *registra el evento de la autenticación del usuario.*
Postcondición	*Ninguna*	
Frecuencia	*100 veces/día*	
Importancia	*Vital*	
Urgencia	*Inmediatamente*	
Estabilidad	*Alta*	
Comentarios	*Una vez que el usuario se ha autenticado, la aplicación se presentará de forma personalizada para el mismo.* *Para simplificar la codificación entenderemos que una autenticación es correcta siempre que la clave sea "pepe". Cuando se introduzca esta clave sin nombre de usuario se tomará* Anonymous *como nombre de usuario.*	

Además se tienen los distintos requisitos (no) funcionales:

FRQ–001	**Archivo de registro**
Versión	1.0 (Abril / 2005)
Autores	• David Ruiz (Universidad de Sevilla)
Descripción	El sistema deberá registrar la actividad de la aplicación en un archivo de texto en el que cada evento irá precedido de la fecha y la hora a la que ocurre.

NFR–001	**Acceso a la base de datos**
Versión	1.0 (Abril / 2005)
Autores	• David Ruiz (Universidad de Sevilla)
Descripción	El sistema deberá minimizar el número de accesos a la base de datos, ya que existen problemas de rendimiento en el servidor de bases de datos.

Con estos requisitos, el PD *Singleton* ayuda a garantizar que:

- Existe un único objeto que almacena la información del usuario que trabaja con la aplicación (nombre de usuario y *password*). Dicho objeto se

puede utilizar para personalizar la aplicación en función al nombre del usuario.

- Existe un único objeto que escribe en el archivo de registro en exclusión mutua con el resto de objetos del sistema.

1.3 Desarrollo de la práctica

El diagrama de clases de la práctica es el que se muestra en el Figura 1.2. Empezaremos por la clase que implementa el objeto encargado de escribir en el fichero de *log*.

Cree el paquete isw2.lab3.traverv0 y añada la clase Tracer. Las variables privadas y el constructor de esta clase son los siguientes:

```
private static Tracer instance;
private PrintWriter out;
private DateFormat dt;
private static final String filename="log.txt";

private Tracer(){
    try{
    out = new PrintWriter(
     new BufferedOutputStream( new FileOutputStream(filename,true)),true);
    dt = DateFormat.getDateTimeInstance(
        DateFormat.SHORT, // date style
        DateFormat.DEFAULT, // time style
        new Locale("es","ES")); // localisation (language,country)
    }
    catch(Exception e){
        e.printStackTrace();
    }
}
```

Fíjese que el constructor de esta clase es privado y que es responsabilidad del método getInstance() devolver el único objeto que se puede crear de la misma. Los métodos para añadir una nueva línea al archivo de *log* y devolver el contenido del mismo son los siguientes:

```
synchronized public void addLine(String msg){
    out.print(dt.format(new Date()));
    out.println(": " + msg);
}
public BufferedReader getContent(){
    BufferedReader result=null;
```

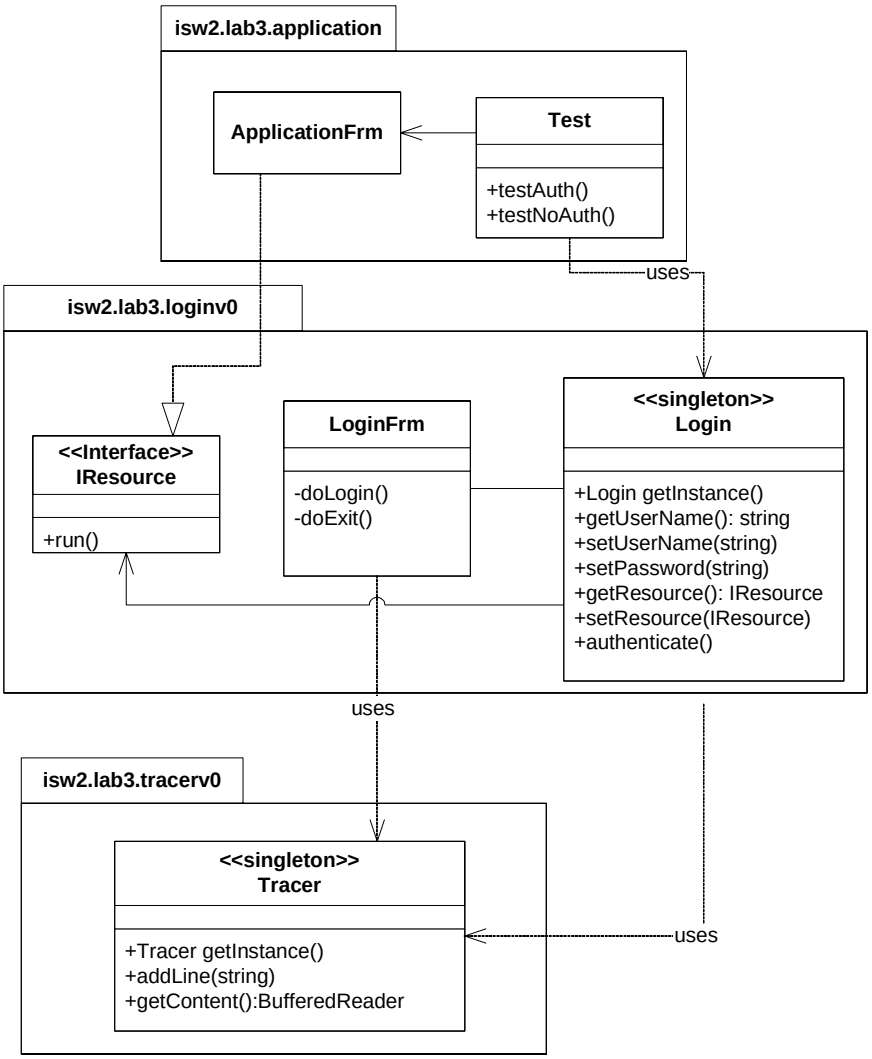


Figura 1.2: *Vista lógica de las capas de la aplicación.*

```
        try{
            result = new BufferedReader(new FileReader(filename));
        }
        catch(Exception e){
            e.printStackTrace();
        }
        return result;
    }
```

Fíjese que el método addLine es synchronized, por lo que garantiza que sólo un hilo del sistema puede estar ejecutando su código simultáneamente. Asimismo, el método getContent() no es synchronized, por lo que se podrá leer el archivo mientras se añade una línea.

El siguiente paso es crear un paquete isw2.lab3.loginv0 y añadir la interface IResource, la clase Login y la clase visual LoginFrm que extenderá a Jframe. El primer interfaz contendrá un único método run(), ese método será el que se ejecute cuando la autenticación tenga éxito.

Para la clase Login, las variables privadas y el constructor son los siguientes:

```
    private String username;
    private String password;
    private static Login instance;
    private static IResource resource;
    private static final String masterPassword = "pepe";

    private Login(){
        username="";
        password="";
    }
```

Fíjese que la única forma de obtener la referencia al único objeto que puede existir de Tracer es invocando a su método estático getInstance().

Los métodos get y set los utilizaremos para leer y escribir, respectivamente, las distintas propiedades. A modo de ejemplo el código que se encarga de leer y escribir la propiedad Username es el siguiente:

```
    public String getUsername(){
        String result;
        if (username.equals("")){
            username = "Anonymous";
        }
        result = username;
        return result;
    }
```

```
public void setUsername(String u){
    username = u;
}
```

Para autenticar a los usuarios vamos a mostrar la ventana de la Figura 1.1.(a), delegando esta tarea en un método privado validate() que será invocado cuando se pulse el botón de *Login!* de la ventana anterior. Este método será el encargado de escribir en el archivo de *log*, quedando:

```
public void authenticate(){
    (new LoginFrm()).show();
}
boolean validate(){
    boolean result = false;

    if (password.compareTo(masterPassword) == 0) {
        Tracer.getInstace().addLine(getUsername() + " logged");
        result = true;
    }
    else {
        Tracer.getInstace().addLine(getUsername() + " refused");
    }
    return result;
}
```

Fíjese que esta implementación permite el acceso a cualquier usuario cuya contraseña sea masterPassword, sin que afecte el nombre del usuario en la aplicación.

Para terminar con este paquete falta la clase visual LoginFrm. Lo primero que hay que hacer una vez que hemos añadido la clase visual al paquete es darle el aspecto de la Figura 1.1.(a). Al JTextField que captura el nombre del usuario le llamaremos usernameTxt, al JPasswordField le llamaremos passwordTxt y a los botones de "Login!" y "Exit" le llamaremos loginBtn y exitBtn, respectivamente. El código asociado al evento actionPerformed lo escribiremos en sendos métodos privados doLogin y doExit, quedando:

```
private void doLogin(){
    login.setUsername(userNameTxt.getText());
    login.setPassWord(new String(passwordTxt.getPassword()));
    if (login.validate()){
        login.getResource().run();
        setVisible(false);
    }
}
private void doExit(){
```

```
        System.exit(0);
    }
```

Fíjese que login es la referencia al objeto que guarda la información del usuario. Cuando se pulsa el botón de "Login!" se le da valor a este objeto y se invoca al método privado validate(). En esta primera práctica, cuando falla la autenticación no vamos hacer nada. La política a seguir cuando se produce un fallo (finalizar la aplicación, limitar el número de fallos, etc) vamos a tratarla en una práctica posterior.

Para terminar ya sólo faltan las clases del paquete isw2.lab3.application. En éste vamos a tener 3 clases. La primera de ellas es una clase visual llamada ApplicationFrm que extenderá a JFrame e implementará el interfaz IResource, su aspecto es el de la Figura 1.1.(b), donde el texto "Anonymous" es un JLabel, al que llamaremos userNameLbl, que cuando se carga la ventana muestra el nombre del usuario que ha iniciado la sesión (esta información la obtendrá del único objeto de la clase Login que puede existir en el sistema). De la misma forma se tiene un control JTextArea, al que llamaremos logTxt, que cuando se carga la aplicación muestra el contenido del archivo de *log*. En el archivo de *log* se tiene que añadir una entrada para registrar que la aplicación se inicia y se cierra. De esta forma, los manejadores de los eventos windowOpened y windowClosing son, respectivamente:

```
    private void doWindowOpened(){
        userNameLbl.setText(Login.getInstance().getUsername());
        Tracer.getInstace().addLine(Login.getInstance().getUsername()+
            " has started the application");
        try {
            logTxt.read(Tracer.getInstace().getContent(),null);
        }
        catch (IOException e1) {
            e1.printStackTrace();
        }
    }
    private void doWindowClosing(){
        Tracer.getInstace().addLine(Login.getInstance().getUsername()+
            " has closed the application");
        System.exit(0);
    }
```

Esta clase tiene que implementar el método run(), que es lo que se ejecuta cuando la autenticación tiene éxito. En este caso, mostrar la ventana de la aplicación, es decir:

```
    public void run(){
        this.show();
```

```
}
```

La siguientes clases son Test y Main, la primera contendrá un método testAuth() para ejecutar la aplicación con autenticación y un método testNoAuth() para ejecutarla sin autenticación. Main es el punto de entrada a la aplicación, cuyo objetivo es crear un objeto de tipo Test y llamar a sus métodos, por ejemplo:

```
public class Main {
    public static void main(String args) {
        Test t = new Test();
        t.testAuth();
        //t.testNoAuth();
    }
}
...
public class Test {

    Test(){ }

    void testAuth()
    {
        Login.getInstance().setResource(new ApplicationFrm());
        Login.getInstance().authenticate();
    }

    void testNoAuth()
    {
        (new ApplicationFrm()).show();
    }
}
```

Fíjese que para ejecutar la aplicación sin autenticación basta con llamar el método show() del JFrame (también valdría llamar al método run() de IResource) que contiene a la aplicación. Por el contrario, si queremos autenticación tenemos que obtener la referencia al objeto de Login, darle valor a su propiedad Resource con el objeto que implementa el método run() y llamar a su método authenticate().

1.4 Ejercicios

► **CUESTIÓN 1.1:** ¿Qué pasaría si quitásemos el modificador static en el método getInstance()?

► **CUESTIÓN 1.2:** ¿Y el modificador synchronized del método addLine()?

► **CUESTIÓN 1.3:** ¿Qué efectos tiene en el rendimiento de un método hacerlo synchronized? Haga un test que mida el tiempo que tardan 1000 hilos en ejecutar 10000 invocaciones de un método síncrono frente a uno que no lo es.

1.5 Solución

Los fuentes completos de esta práctica estarán disponibles en la página Web de la asignatura cuando se imparta en el laboratorio.