INFORMÁTICA Y MULTIMEDIA
TÉCNICAS DE DESARROLLO DE SOFTWARE
Consultores: Joan Esteve, Fatos Xhafa, Jorge Mendoza

Desarrollo de una aplicación Java con internacionalización I18N

El objetivo de este enunciado es realizar una pequeña práctica antes de empezar la etapa de implementación del proyecto TDS, situada en las prácticas PEC3 y PEC4 de evaluación continuada.

Consiste en aplicar una *best practice* en la implementación de interfícies gráficas de usuario (GUI) mediante Java: usar **el servicio de internacionalización I18N** toda la gestión de mensajes, etiquetas, y textos mostrados en la aplicación cliente.

La importancia de usar I18N viene, originalmente, relacionada en la gestión de los posibles idiomas en los que se puede ofrecer dicha aplicación, y la importancia de eliminar del código fuente todos los mensajes que se quieran presentar al usuario.

Pensemos, por un momento, que decidiéramos construir una aplicación Java 6, con una interfície gráfica basada en el paquete **Swing**, muy similar a la que implementaréis durante el proyecto de TDS.

¿Que pasaría si pusiéramos, directamente, todos los mensajes y textos de la aplicación dentro del código fuente de nuestras clases? Para crear, por ejemplo, un botón con una etiqueta “Añadir usuario” implementaríamos la siguiente línea de código:

```
JButton button = new JButton("Añadir usuario");
```

Esto cumpliría nuestro objetivo: la etiqueta del botón creado sería “Añadir usuario”:

Pero, ¿que sucede si en cualquier momento quisiéramos cambiar esta etiqueta por “Añadir nuevo usuario”? Tendríamos que modificar dicho texto en el código fuente de la clase, y volver a recompilar la aplicación (no siempre posible en un contexto real).

```
JButton button = new JButton("Añadir nuevo usuario");
```

Lo mismo sucedería en el caso que, a partir de un determinado, momento quisiéramos ofrecer la posibilidad de tener diferentes idiomas en la aplicación. (Este no va a ser el caso de TDS, donde solo se requiere implementar la aplicación en un idioma, pero podría darse esta situación a un proyecto real)

La solución es delegar en una clase Java la gestión de textos que se mostrarán en la aplicación: etiquetas, botones, paneles, títulos, ventanas informativas, etc...

En el caso de logs, o mensajes de error internos de la aplicación que no se muestran al usuario final, no es necesario aplicar este tratamiento. Pues su objetivo es más bien que sean comprensibles para el programador, y no para el cliente final.

La herramienta **I18N** es una buena solución para este caso. La internacionalización es el proceso de diseñar e implementar una aplicación adaptable a diferentes idiomas y regiones, sin necesidad de recompilar todo el código fuente en cada cambio.

Es una manera de desacoplar la lógica de implementación de la lógica de mensajes de información y etiquetas que se presentan al usuario. De esta forma, se tiene separado por un lado el código de la aplicación. Y por otro, sus elementos textuales se guardan en diferentes ficheros de texto plano, fácilmente manejables mediante identificadores, y con la posibilidad de mantener dichos mensajes sin hacer modificaciones en la aplicación: usando procesadores de texto, traductores, correctores ortográficos, etc...

Tenéis muchos detalles, en los enlaces de bibliografía adjuntos al documento, sobre cual es la mejor manera de gestionar los mensajes de una aplicación sin tener que ponerlos *hard-coded* (esto es, escribirlos directamente en el código fuente Java).

Lo que se pide, en la asignatura de TDS, es que vuestra aplicación del proyecto **NO** tenga ningún mensaje o texto de usuario puesto *hard-coded* en el código fuente. No se puede permitir esta mala práctica de ninguna manera, en una aplicación gráfica cliente-servidor como la que vais a desarrollar.

Para ello, el objetivo de este enunciado es enseñaros una manera de ofrecer un servicio de internacionalización a vuestra aplicación, que a partir de ahora llamaremos **Gestor de Mensajes**.

Os vamos a dar un ejemplo implementado de un objeto que actúa como Gestor de Mensajes, con sencillos métodos para leer mensajes de un fichero de texto plano creado por vosotros. Podéis utilizar esta clase en vuestras aplicaciones de la PEC3 y la PEC4, o desarrollar vosotros mismos vuestro propio gestor de mensajes. Lo importante es que el proyecto entregado a los consultores de la asignatura no tenga los textos puestos directamente en el código fuente.

Si queréis programar vuestro propio gestor, las clases más importantes que tenéis que considerar son las siguientes:

```
java.util.Locale
java.util.ResourceBundle
```

El gestor de mensajes que os ofrecemos es una clase llamada `TDSLanguageUtils`, definida en el *package* `edu.uoc.tds.i18n.*`.

Por tanto, cualquier nueva clase que implementéis vosotros y que quiera utilizar este gestor deberá importarlo en su cabecera.

Ejemplo de uso del gestor de mensajes

Adjunto a este manual tenéis la documentación, en formato **javadoc**, y el código fuente de la clase `TDSLanguageUtils`. De momento, vamos a mostrar un ejemplo de como usar el gestor. Sus funciones más importantes son:

```
public static boolean setDefaultLanguage(java.lang.String baseName)
```

Indica al gestor de mensajes cual es la ubicación de los ficheros de idiomas. El idioma que se aplica, en este caso, es el que viene por defecto en vuestra máquina virtual, que será seguramente el castellano. Ésta es la única función que realmente necesitáis para TDS, pues no os pedimos multi-idioma en vuestra aplicación. Sólo que los mensajes se lean del fichero de texto que hayáis definido.

Si vuestro fichero de mensajes es `messages.properties`, y lo tenéis en la carpeta `i18n` del proyecto, debéis pasar como parámetro `baseName` el valor "i18n/messages".

A parte, aunque no es necesario para TDS, también tenéis una función para añadir la posibilidad de cambiar el idioma de la aplicación, definiendo varios ficheros de texto.

```
public static boolean setLanguage(java.lang.String baseName,
                                  java.util.Locale locale)
```

Indica al gestor la ubicación de los ficheros con los idiomas, y el idioma que queréis que utilice. Los dos idiomas más usados que veréis son el castellano ("es_ES"), el catalán ("ca_ES") y el inglés ("en").

Para que vuestra aplicación tuviera esos idiomas, deberíais tener en la carpeta `i18n` los ficheros `messages_es_ES.properties`, `messages_ca_ES.properties`, y `messages_en.properties`. Y pasaríais a la función *setLanguage* el valor "i18n/messages", y el idioma a usar, en formato `java.util.Locale`.

Finalmente, una vez configurado fichero e idioma, para leer un texto tenéis la función:

```
public static java.lang.String getMessage(java.lang.String key)
```

Que devuelve el mensaje con el identificador `key`, obtenido del fichero de idiomas que le hayáis configurado al Gestor de Mensajes.

Observad que todos los métodos de la clase son de tipo estático. Eso indica que `TDSLanguageUtils` actúa como un *singleton*, es decir, hay una única clase gestora de mensajes para todo el código de vuestra aplicación. Y no es necesario instanciar un nuevo gestor de mensajes cada vez que tengáis que leer un texto en una clase.

Vamos a hacer el ejemplo anterior del botón "*Añadir usuario*", pero utilizando el gestor de mensajes. No lo hacemos multi-idioma, el idioma por defecto será castellano.

Primero de todo, definimos el archivo `i18n/messages.properties`. Y le añadimos una entrada con el identificador "boton1.etiqueta":

```
boton1.etiqueta=Añadir usuario
```

A continuación, en nuestro código, cargamos el gestor de mensajes con este fichero de idiomas, y le indicamos que utilice el idioma por defecto.

```
TDSLanguageUtils.setDefaultLanguage("i18n/messages");
```

Por último, creamos el botón (que es un componente `javax.swing.JButton`), y le decimos que su etiqueta será el texto rescatado con el gestor de mensajes:

```
String etiqueta = TDSLanguageUtils.getMessage("boton1.etiqueta");
JButton button = new JButton(etiqueta);
```

De esta manera, tendremos el mismo botón que antes, pero el texto "*Añadir usuario*" de su etiqueta no se ha metido en el código, si no separado en el fichero anterior. Si hubiéramos querido modificar la etiqueta del botón por "*Añadir nuevo usuario*", tan solo habría sido necesario tocar este texto el fichero de idiomas, sin tener que recompilar la clase ni la aplicación por completo.

Además, si hubiéramos querido sacar la aplicación en otro idioma, por ejemplo inglés, habría sido necesario tener otro fichero `i18n/messages_en.properties`, con la traducción deseada en este idioma:

```
boton1.etiqueta=Add new user
```

Y avisar al gestor de mensajes que tiene que usar otro idioma:

```
TDSLanguageUtils.setLanguage("i18n/messages", new Locale("en"));
String etiqueta = TDSLanguageUtils.getMessage("boton1.etiqueta");
JButton button = new JButton(etiqueta);
```

Ahora el botón ahora mostraría su etiqueta en inglés.

Ejemplo completo gestor de mensajes

Finalizada esta parte teórica, lo que pedimos en TDS es que ningún mensaje de vuestra interfície gráfica tenga código *hard-coded*. Por favor, tened en cuenta estas indicaciones de cara a la implementación

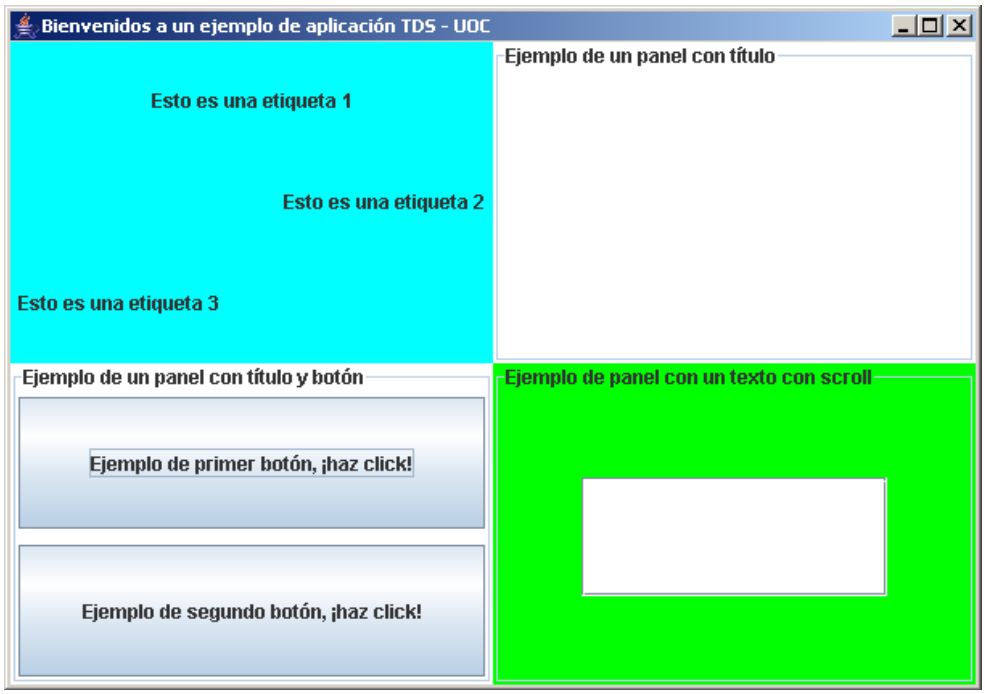
Os vamos a dar un ejemplo de una aplicación que utiliza el gestor de mensajes para mostrar diferentes paneles con botones, etiquetas, áreas de texto. Es una clase `Test`. Observad que el método `main()` de esta clase permite arrancar con el lenguaje por defecto (llamada sin parámetros) o con un idioma (pasado como primer parámetro).

```
public static void main(String args[]) {
	if(args.length==0) {
		TDSLanguageUtils.setDefaultLanguage("i18n/messages");
	}
	if(args.length==1) {
		Locale locale = new Locale(args[0]);
		TDSLanguageUtils.setLanguage("i18n/messages", locale);
	}
	new Test();
}
```

Para ejecutar el ejemplo podéis hacer:

```
> java edu.uoc.tds.Test  // arranca en idioma por defecto (castellano)
> java edu.uoc.tds.Test es_ES  // arranca en castellano
> java edu.uoc.tds.Test ca_ES  // arranca en catalán
> java edu.uoc.tds.Test en  // arranca en inglés
```

Según el idioma, tendréis la interfície gráfica de usuario en el idioma deseado:



Probad de cambiar el idioma para ver los efectos en la aplicación.

Bibliografía

http://www.programacion.net/java/tutorial/i18n/

http://java.sun.com/docs/books/tutorial/i18n/index.html

http://java.sun.com/docs/books/tutorial/i18n/resbundle/propfile.html

Implementación de la clase TDSLanguageUtils

Se adjunta la implementación de la clase por si algún estudiante quiere desarrollar su propio gestor de mensajes i18n.

```
package edu.uoc.tds.i18n;

import java.util.Locale;
import java.util.MissingResourceException;
import java.util.ResourceBundle;

/**Técnicas de Desarrollo de Software
 * Universitat Oberta de Catalunya (UOC)
 * @author Joan Esteve Riasol
 */
public class TDSLanguageUtils  {
	private static ResourceBundle resourceBundle;
	private static String currentBaseName;
	private static Locale currentLocale;

	/**Configura el idioma por defecto en el gestor de idiomas TDSLanguageUtils.
	 * @param  baseName  dirección donde se encuentra el fichero de idiomas
	 * @throws NullPointerException si baseName es nulo
	 * @return un booleano indicando si la operación ha ido bien
	 */
	public static synchronized boolean setDefaultLanguage(String baseName) {
		try {
			currentBaseName=baseName;
			Locale locale = Locale.getDefault();
			resourceBundle = ResourceBundle.getBundle(baseName,locale);
			return true;
		}catch(MissingResourceException e){
			e.printStackTrace();
			return false;
		}
	}

	/**Configura el idioma indicado por el usuario en el gestor TDSLanguageUtils
	 * @param baseName dirección donde se encuentra el fichero de idiomas
	 * @param locale idioma que se quiere indicar
	 * @throws NullPointerException si baseName o locale son nulos
	 * @return un booleano indicando si la operación ha ido bien
	 */
	public static synchronized boolean setLanguage(String baseName, Locale locale) {
		try {
			currentBaseName=baseName;
			currentLocale=locale;
			resourceBundle = ResourceBundle.getBundle(baseName,locale);
			return true;
		}catch(MissingResourceException e){
			e.printStackTrace();
			return false;
		}
	}

	/**Obtiene el texto del fichero de idiomas segun la etiqueta solicitada
	 * @param key etiqueta del fichero de idiomas
	 * @param locale idioma que se quiere indicar
	 * @throws NullPointerException si el gestor no ha sido aún configurado
	 * @return un String con la traducción solicitada
	 */
	public static String getMessage(String key)  {
		try {
			return resourceBundle.getString(key);
		}catch(MissingResourceException e) {
			return new String();
		}
	}

	public static String getCurrentBaseName() {
		return currentBaseName;
	}
	public static Locale getCurrentLocale() {
		return currentLocale;
	}
}
```