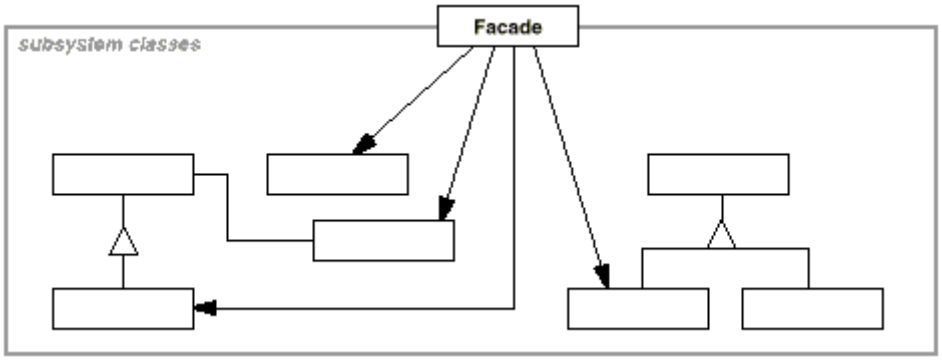
Facade

Diagrama de clases e interfaces

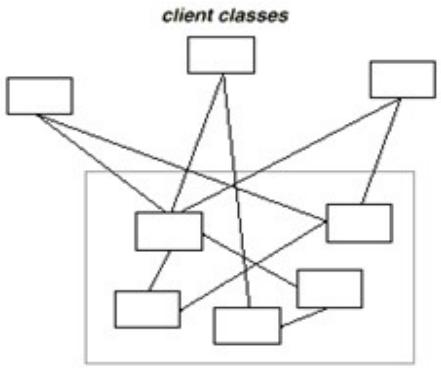


Intención

- El patrón fachada se utiliza para proporcionar una interfaz unificada de alto nivel para un conjunto de clases en un subsistema, haciéndolo así más fácil de usar al código cliente.
- Simplifica el acceso a dicho conjunto de clases, ya que el cliente sólo se comunica con ellas a través de una única interfaz.

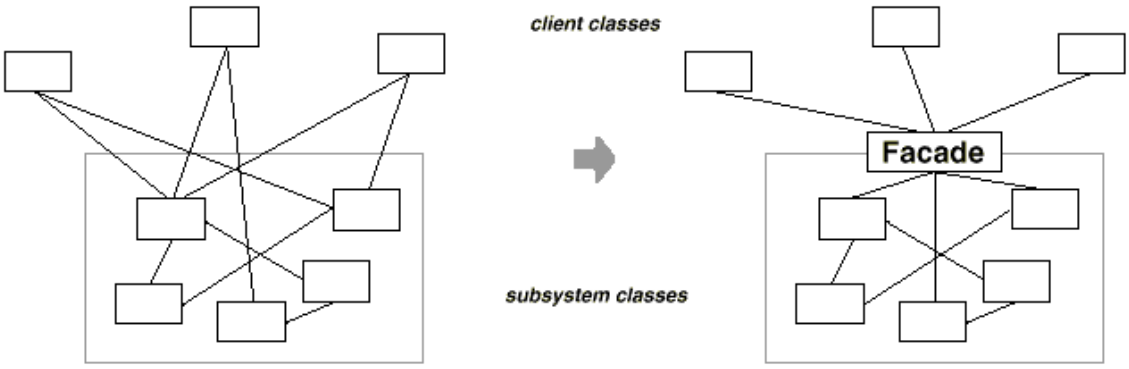
Motivación

Estructurar un sistema por módulos (subsistemas) ayuda a reducir su complejidad. Un subsistema consta de uno o varios grupos de clases, o grupos de clases y otros subsistemas, cuyo objetivo es proporcionar una funcionalidad. No obstante, la interfaz que exponen las clases en un subsistema o conjunto de subsistemas puede llegar a resultar muy compleja, tal y como muestra la siguiente figura:



Un objetivo común de diseño es reducir al mínimo la comunicación y las dependencias entre los subsistemas. Una forma de lograr este objetivo es introducir un objeto fachada que proporcione una interfaz única y simplificada con las operaciones más interesantes de los subsistemas. Estas operaciones deben proporcionar servicios de alto nivel, con un alto nivel de abstracción, ocultando los detalles de implementación del subsistema.

La siguiente figura muestra, a grandes rasgos, cómo se puede aplicar el patrón Facade para simplificar la programación del código cliente sin perder la funcionalidad de los diversos módulos de una aplicación:



Notad que ahora, gracias a la fachada, cualquier cambio en la signatura de un método del subsistema, o incluso el reemplazo de una clase o módulo, no tendrá repercusión alguna en el código cliente, pues la fachada seguirá ofreciendo la misma interfaz. Lógicamente, la fachada sí tendrá que ajustarse de manera adecuada a ese cambio.

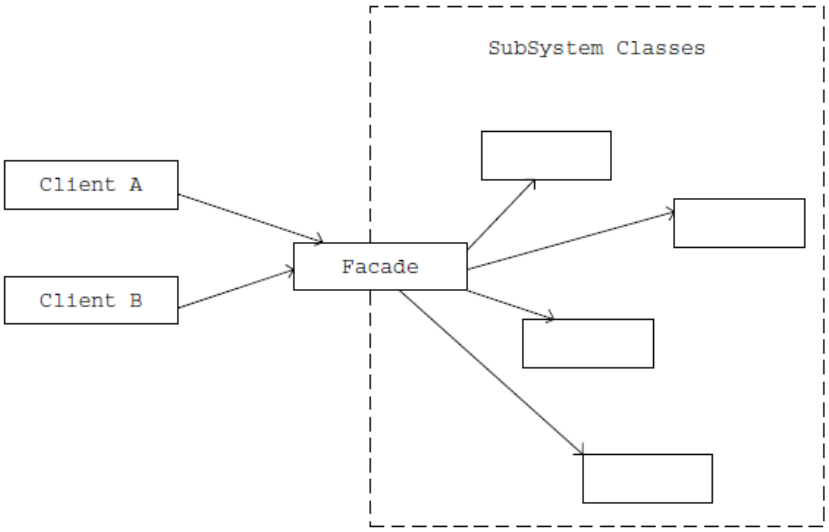
Por tanto, la fachada desacopla al código cliente de las clases del subsistema y lo protege de sus modificaciones, ya que el código cliente sólo se relaciona con la fachada.

Por otro lado, gracias a la fachada, el subsistema es más manejable y sencillo de utilizar para el código cliente, ya que ahora tan sólo se tiene que acceder a la fachada.

No obstante, no estamos ante una cuestión de todo o nada, en el sentido de que si una clase cliente necesita acceder a un servicio de bajo nivel que no proporciona la fachada, será perfectamente válido que acceda directamente al componente del subsistema que ofrece una determinada clase o componente.

Implementación

El funcionamiento del patrón es el mostrado en el diagrama de clases siguiente:



Las clases participantes en el patrón son las siguientes:

- **Fachada (Facade):** Ofrece al código cliente una serie de servicios de alto nivel, consiguiendo de este modo que el uso del subsistema sea una tarea trivial. Sabe qué clases del subsistema son responsables de llevar a cabo una determinada petición del código cliente, por lo que delega en ellas su realización.
- **Clases del subsistema:** implementan la funcionalidad del subsistema. Realizan el trabajo solicitado por la fachada. No conocen la existencia de la fachada ni de las clases cliente.

Aplicabilidad y Ejemplos

El uso del patrón Facade es adecuado cuando:

- Se quiere proporcionar una interfaz sencilla a un complejo subsistema. Esto hace que el subsistema sea fácil de utilizar por el código cliente.
- Se debe evitar el acoplamiento entre el código cliente y el subsistema. La introducción de un componente fachada desacopla ambas partes, fomentando la independencia y portabilidad del subsistema.

- Se quiere descomponer una aplicación por capas. Por ejemplo, capa de presentación, de servicios de negocio y de servicios técnicos. En este caso, una fachada puede ser el punto de entrada a cada capa. De este modo, la comunicación entre capas se realiza a través de la fachada que cada una proporciona, lo cual resulta en un sistema con pocas dependencias.

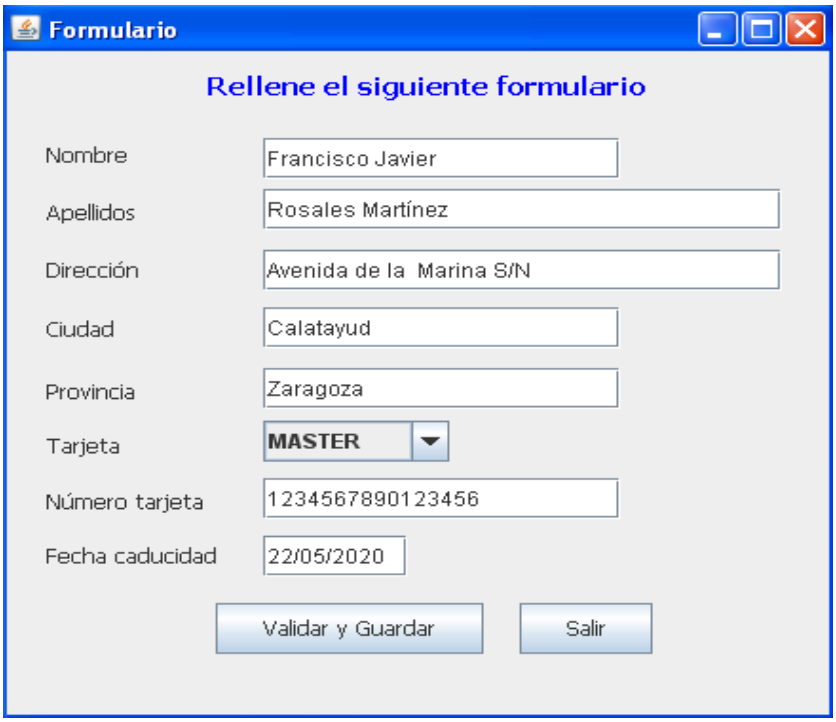
Ejemplo 1: Simplificación de acceso a un subsistema

Supongamos que tenemos que crear una aplicación que permite a los usuarios comprar algún tipo de productos. Uno de los pasos en el proceso de compra consistirá en recoger información perteneciente a los clientes.

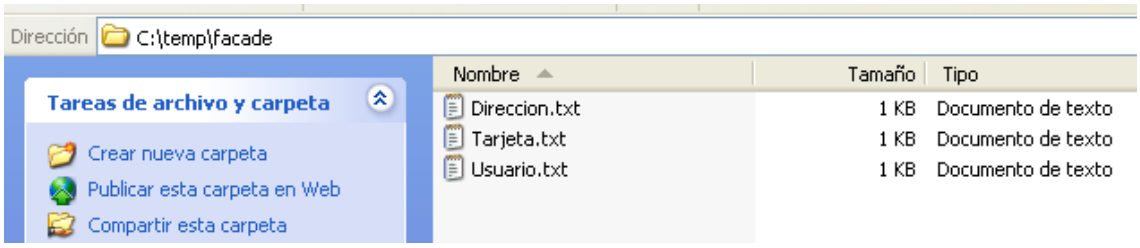
A saber:

- Datos sobre la identidad del usuario.
- Datos relativos a la dirección de envío de los productos.
- Datos sobre la tarjeta de crédito/débito con la que pagará la compra.

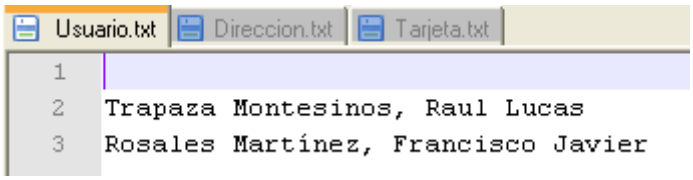
El cliente deberá cumplimentar el siguiente formulario:

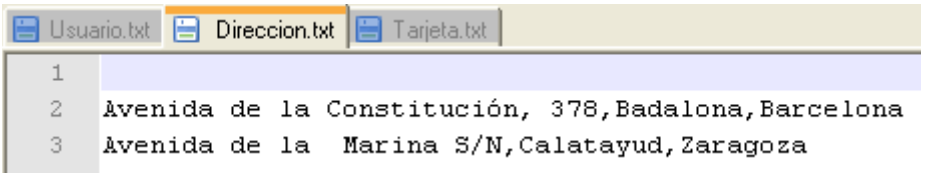


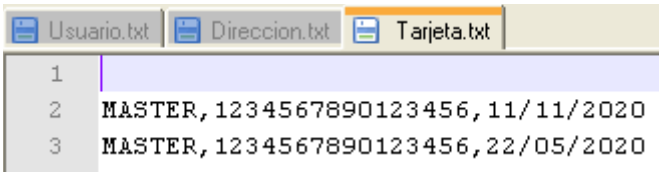
A continuación, pulsará el botón ‘Validar y Guardar’ o ‘Salir’. El botón ‘Validar y Guardar’ almacenará la información en disco, en tres ficheros de texto distintos. El botón ‘Salir’ terminará el programa sin salvar la información:



Estos ficheros sirven de base de datos simple, en tanto que van acumulando toda la información introducida por los diferentes usuarios del programa:



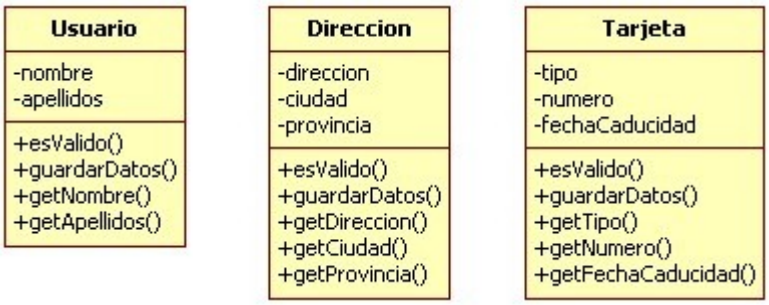




Diseño del programa

El subsistema

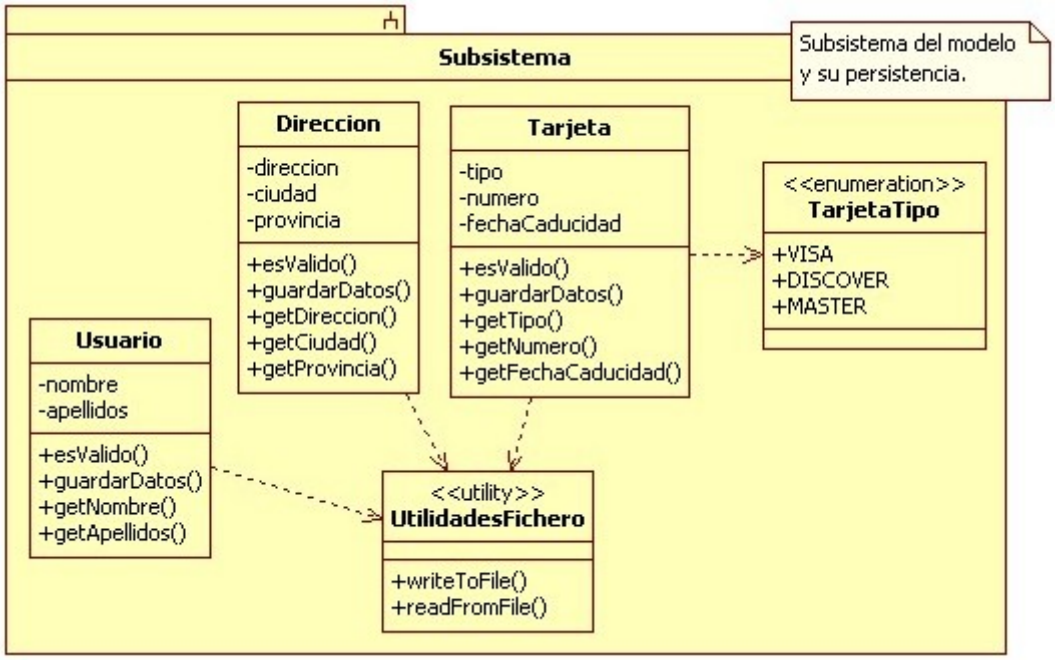
Vamos a crear tres clases para encapsular la información relativa a los clientes: Usuario, Direccion y Tarjeta, cada una con sus propios métodos para validar y salvar sus respectivos datos.



También tendremos una enum llamada TarjetaTipo, que contendrá algunas definiciones para la clase Tarjeta.

Como apoyo, estas clases colaborarán con una clase llamada UtilidadesFichero que hará el trabajo a bajo nivel relacionado con la lectura y escritura de ficheros.

Todas estas clases forman un subsistema: *el modelo y su persistencia*, tal y como muestra la figura siguiente:



Veamos el código fuente del subsistema:

Usuario.java

```
package estructurales.facade.subsistema;

public class Usuario {
	private String nombre;
	private String apellidos;
```

```
	private final String FICHERO_USUARIO =
"c://temp//facade//Usuario.txt";
	private final boolean APPEND = true;
	private final boolean CARACTER_NUEVA_LINEA = true;

	public Usuario(String nom, String apel) {
		nombre = nom;
		apellidos = apel;
	}

	public String getNombre() { return nombre; }
	public String getApellidos() { return apellidos; }

	/*
	 * Hacemos una comprobación simple para no complicar
	 * el ejemplo. El apellido tiene que tener dos o más
	 * caracteres para ser valido.
	 */
	public boolean esValido() {
		if (getApellidos().trim().length() < 2)
			return false;

		return true;
	}

	/*
	 * Guardar los datos del usuario en un fichero de texto
	 */
	public boolean guardarDatos() {
		UtilidadesFichero fUtil = new UtilidadesFichero();
		String lineaDatos = getApellidos() + ", " + getNombre();
		return fUtil.writeToFile(FICHERO_USUARIO,
				lineaDatos, APPEND, CARACTER_NUEVA_LINEA);
	}

}
```

Direccion.java

```
package estructurales.facade.subsistema;

public class Direccion {
	private String direccion;
	private String ciudad;
	private String provincia;
	private final String FICHERO_USUARIO =
"c://temp//facade//Direccion.txt";
	private final boolean APPEND = true;
	private final boolean CARACTER_NUEVA_LINEA = true;

	public Direccion(String dir, String ciu, String prov) {
		direccion = dir;
		ciudad = ciu;
		provincia = prov;
	}

	public String getDireccion() { return direccion; }
	public String getCiudad() { return ciudad; }
```

```
	public String getProvincia() { return provincia; }

	/*
	 * Hacemos una comprobación simple para no complicar
	 * el ejemplo. La provincia tiene que tener dos o más
	 * caracteres para ser valida.
	 */
	public boolean esValido() {
		if (getProvincia().trim().length() < 2)
			return false;

		return true;
	}

	/*
	 * Guardar los datos de la direccion en un fichero de texto
	 */
	public boolean guardarDatos() {
		UtilidadesFichero fUtil = new UtilidadesFichero();
		String dataLine = getDireccion() + "," + getCiudad() + ","
+ getProvincia();
		return fUtil.writeToFile(FICHERO_USUARIO,
				dataLine, APPEND, CARACTER_NUEVA_LINEA);
	}

}
```

Tarjeta.java

```
package estructurales.facade.subsistema;

public class Tarjeta {
	private TarjetaTipo tipo;
	private String numero;
	private String fechaCaducidad;
	private final String FICHERO_TARJETA =
"c://temp//facade//Tarjeta.txt";
	private final boolean APPEND = true;
	private final boolean CARACTER_NUEVA_LINEA = true;

	public Tarjeta(TarjetaTipo tip, String num, String fechaCad) {
		tipo = tip;
		numero = num;
		fechaCaducidad = fechaCad;
	}

	public TarjetaTipo getTipo() { return tipo; }
	public String getNumero() {  return numero; }
	public String getFechaCaducidad() { return fechaCaducidad; }

	/*
	 * Hacemos una comprobación simple para no complicar
	 * el ejemplo. Cada tipo de tajeta debe tener una
	 * longitud de caracteres específica.
	 */
	public boolean esValido() {
		if (getTipo().equals(TarjetaTipo.VISA)) {
```

```
                  return (getNumero().trim().length() ==
TarjetaTipo.LENGHT_VISA);
            }
            if (getTipo().equals(TarjetaTipo.DISCOVER)) {
                  return (getNumero().trim().length() ==
TarjetaTipo.LENGHT_DISCOVER);
            }
            if (getTipo().equals(TarjetaTipo.MASTER)) {
                  return (getNumero().trim().length() ==
TarjetaTipo.LENGHT_MASTER);
            }

            return false;
      }

      public boolean guardarDatos() {
            UtilidadesFichero fUtil = new UtilidadesFichero();
            String dataLine = getTipo() + "," + getNumero() + ","
                        + getFechaCaducidad();
            return fUtil.writeToFile(FICHERO_TARJETA,
                        dataLine, APPEND, CARACTER_NUEVA_LINEA);
      }

}
```

TarjetaTipo.java

```
package estructurales.facade.subsistema;

public enum TarjetaTipo {
      VISA, DISCOVER, MASTER;
      public static final int LENGHT_VISA = 14;
      public static final int LENGHT_DISCOVER = 15;
      public static final int LENGHT_MASTER = 16;
}
```

UtilidadesFichero.java

```
package estructurales.facade.subsistema;

import java.io.*;
import java.util.*;

/*
 * Clase de utilidad para leer y escribir datos
 * mediante ficheros de texto.
 */
public class UtilidadesFichero {

      DataOutputStream dos;

      /*
       * Escribir texto en un fichero
       */
      public boolean writeToFile(String fileName,
```

```
                  String dataLine, boolean isAppendMode, boolean
isNewLine)
      {
            if (isNewLine) {
                  dataLine = "\n" + dataLine;
            }

            try {
                  File outFile = new File(fileName);
                  if (isAppendMode) {
                        dos = new DataOutputStream(new
FileOutputStream(fileName, true));
                  } else {
                        dos = new DataOutputStream(new
FileOutputStream(outFile));
                  }

                  dos.writeBytes(dataLine);
                  dos.close();
            } catch (FileNotFoundException ex) {
                  return false;
            } catch (IOException ex) {
                  return false;
            }
            return true;

      }

      /*
       * Leer con buffer datos de un fichero
       */
      public String readFromFile(String fileName) {
            String DataLine = "";
            try {
                  File inFile = new File(fileName);
                  BufferedReader br =
                        new BufferedReader(
                                    new InputStreamReader(
                                                new
FileInputStream(inFile)));

                  DataLine = br.readLine();
                  br.close();
            } catch (FileNotFoundException ex) {
                  return null;
            } catch (IOException ex) {
                  return null;
            }
            return (DataLine);

      }

      public boolean isFileExists(String fileName) {
            File file = new File(fileName);
            return file.exists();
      }

      public boolean deleteFile(String fileName) {
            File file = new File(fileName);
```

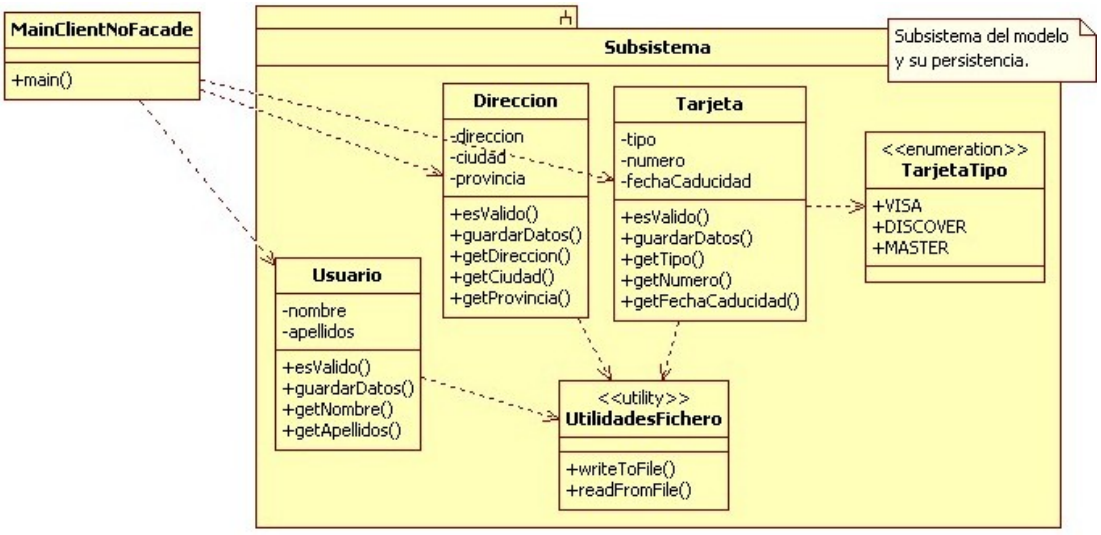
```
            return file.delete();
      }

      /*
       * Leer datos de un fichero y almacenar cada fila
       * como elemento de un vector.
       */
      public Vector<String> fileToVector(String fileName) {
            Vector<String> v = new Vector<String>();
            String inputLine;
            try {
                  File inFile = new File(fileName);
                  BufferedReader br =
                        new BufferedReader(
                                    new InputStreamReader(
                                                new
FileInputStream(inFile)));

                  while ((inputLine = br.readLine()) != null) {
                        v.addElement(inputLine.trim());
                  }
                  br.close();
            } catch (FileNotFoundException ex) {
                  ex.printStackTrace();
            } catch (IOException ex) {
                  ex.printStackTrace();
            }
            return v;
      }

      // Leer los elementos de un vector y guardarlos en un fichero
      public void vectorToFile(Vector<Object> elementos, String
fileName) {

            for (Object o : elementos) {
                  writeToFile(fileName, String.valueOf(o), true, true);
            }
      }

}
```

El código cliente

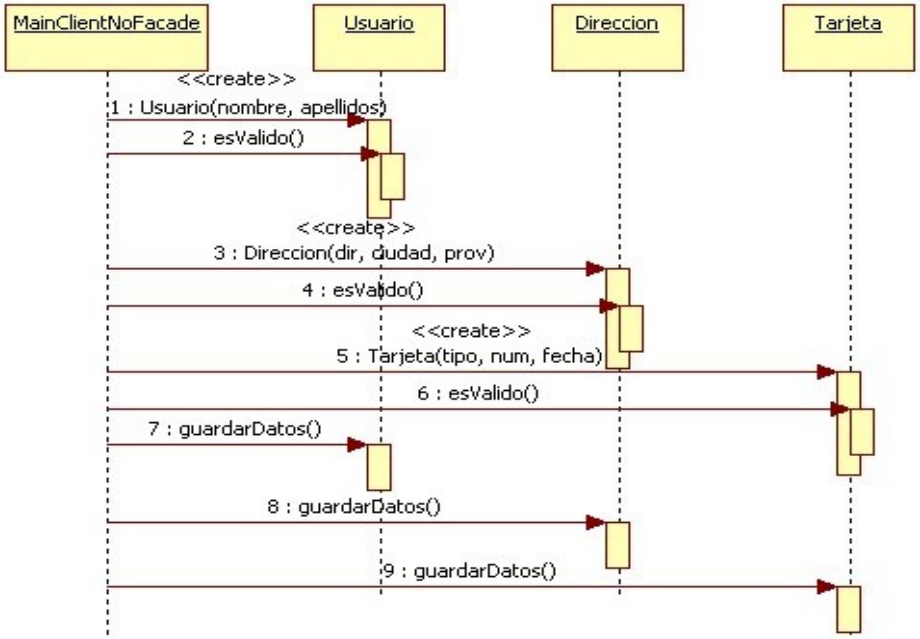
Por otro lado, tendremos una clase cliente encargada de mostrar el formulario e interactuar con los usuarios para obtener la información. A esta clase la llamaremos MainClientNoFacade, ya que no utilizaremos el patrón Facade en esta primera aproximación.

Cuando el usuario quiera salvar los datos, la clase MainClientNoFacade realizará lo siguiente:

- Creará un objeto de cada clase: Usuario, Direccion y Tarjeta. Los respectivos métodos constructores permiten que cada objeto sea inicializado justo en el momento de su creación.
- Comprobará que los datos introducidos en el formulario son válidos. Para ello utilizará el método isValido() de los objetos anteriores.
- Salvará los datos utilizando el método guardarDatos() de estos objetos.



El diagrama de la figura siguiente muestra el flujo de mensajes entre los objetos.



Aplicando el patrón Facade se obtendría un mejor diseño, ya que habría un acoplamiento mucho menor entre la clase cliente y el subsistema. Esto lo veremos posteriormente.

Veamos el código para el programa principal:

MainClientNoFacade.java

```
package estructurales.facade.client;
import java.awt.EventQueue;

public class MainClientNoFacade {

	private JFrame marco;
	private JTextField txtNombre, txtApellidos;
	private JTextField txtDireccion, txtCiudad, txtProvincia;
	private JComboBox cmbTarjeta;
	private JTextField txtNumeroTarjeta, txtFechaCaducidad;
	private JLabel lblMensaje;

	public static void main(String[] args) {
		EventQueue.invokeLater(new Runnable() {
			public void run() {
				try {
					MainClientNoFacade window = new
MainClientNoFacade();
					window.marco.setVisible(true);
				} catch (Exception e) {
					e.printStackTrace();
				}
			}
		});
	}

	public MainClientNoFacade() {
		initialize();
	}

	private void initialize() {
		marco = new JFrame();
		marco.setTitle("Formulario");
		marco.setBounds(100, 100, 441, 438);
		marco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		marco.getContentPane().setLayout(null);

		JLabel lblTitulo = new JLabel("Rellene el siguiente
formulario");
		lblTitulo.setHorizontalAlignment(SwingConstants.CENTER);
		lblTitulo.setForeground(Color.BLUE);
		lblTitulo.setFont(new Font("Tahoma", Font.BOLD, 15));
		lblTitulo.setBounds(10, 11, 422, 20);
		marco.getContentPane().add(lblTitulo);

		JLabel lblNombre = new JLabel("Nombre");
		lblNombre.setFont(new Font("Tahoma", Font.PLAIN, 12));
		lblNombre.setBounds(21, 56, 46, 14);
```

```
                marco.getContentPane().add(lblNombre);

                txtNombre = new JTextField();
                txtNombre.setBounds(135, 53, 188, 25);
                marco.getContentPane().add(txtNombre);
                txtNombre.setColumns(10);

                JLabel lblApellidos = new JLabel("Apellidos");
                lblApellidos.setFont(new Font("Tahoma", Font.PLAIN, 12));
                lblApellidos.setBounds(21, 91, 76, 14);
                marco.getContentPane().add(lblApellidos);

                txtApellidos = new JTextField();
                txtApellidos.setColumns(10);
                txtApellidos.setBounds(135, 84, 273, 25);
                marco.getContentPane().add(txtApellidos);

                JLabel lblDireccin = new JLabel("Direcci\u00F3n");
                lblDireccin.setFont(new Font("Tahoma", Font.PLAIN, 12));
                lblDireccin.setBounds(21, 126, 76, 14);
                marco.getContentPane().add(lblDireccin);

                txtDireccion = new JTextField();
                txtDireccion.setColumns(10);
                txtDireccion.setBounds(135, 121, 273, 25);
                marco.getContentPane().add(txtDireccion);

                JLabel lblCiudad = new JLabel("Ciudad");
                lblCiudad.setFont(new Font("Tahoma", Font.PLAIN, 12));
                lblCiudad.setBounds(21, 162, 76, 14);
                marco.getContentPane().add(lblCiudad);

                txtCiudad = new JTextField();
                txtCiudad.setColumns(10);
                txtCiudad.setBounds(135, 156, 188, 25);
                marco.getContentPane().add(txtCiudad);

                JLabel lblTarjeta = new JLabel("Tarjeta");
                lblTarjeta.setFont(new Font("Tahoma", Font.PLAIN, 12));
                lblTarjeta.setBounds(21, 233, 76, 14);
                marco.getContentPane().add(lblTarjeta);

                cmbTarjeta = new JComboBox();
                cmbTarjeta.addItem(TarjetaTipo.VISA);
                cmbTarjeta.addItem(TarjetaTipo.MASTER);
                cmbTarjeta.addItem(TarjetaTipo.DISCOVER);
                cmbTarjeta.setBounds(135, 225, 98, 25);
                marco.getContentPane().add(cmbTarjeta);

                JLabel lblNumeroTarjeta = new JLabel("N\u00FAmero
tarjeta");
                lblNumeroTarjeta.setFont(new Font("Tahoma", Font.PLAIN,
12));
                lblNumeroTarjeta.setBounds(21, 267, 114, 14);
                marco.getContentPane().add(lblNumeroTarjeta);

                txtNumeroTarjeta = new JTextField();
                txtNumeroTarjeta.setColumns(10);
                txtNumeroTarjeta.setBounds(135, 260, 188, 25);
```

```
            marco.getContentPane().add(txtNumeroTarjeta);

            JLabel lblFechaCaducidad = new JLabel("Fecha caducidad");
            lblFechaCaducidad.setFont(new Font("Tahoma", Font.PLAIN,
12));
            lblFechaCaducidad.setBounds(21, 300, 114, 14);
            marco.getContentPane().add(lblFechaCaducidad);

            txtFechaCaducidad = new JTextField();
            txtFechaCaducidad.setColumns(10);
            txtFechaCaducidad.setBounds(135, 295, 76, 25);
            marco.getContentPane().add(txtFechaCaducidad);


            JLabel lblProvincia = new JLabel("Provincia");
            lblProvincia.setFont(new Font("Tahoma", Font.PLAIN, 12));
            lblProvincia.setBounds(21, 200, 76, 14);
            marco.getContentPane().add(lblProvincia);

            txtProvincia = new JTextField();
            txtProvincia.setColumns(10);
            txtProvincia.setBounds(135, 193, 188, 25);
            marco.getContentPane().add(txtProvincia);

            lblMensaje = new JLabel("");
            lblMensaje.setBounds(0, 379, 432, 25);
            marco.getContentPane().add(lblMensaje);

            JButton btnValidarYGuardar = new JButton("Validar y
Guardar");
            btnValidarYGuardar.addActionListener(new ActionListener() {
                  public void actionPerformed(ActionEvent e) {
                        validarYguardar();
                  }
            });
            btnValidarYGuardar.setFont(new Font("Tahoma", Font.PLAIN,
12));
            btnValidarYGuardar.setBounds(110, 336, 141, 31);
            marco.getContentPane().add(btnValidarYGuardar);

            JButton btnSalir = new JButton("Salir");
            btnSalir.addActionListener(new ActionListener() {
                  public void actionPerformed(ActionEvent e) {
                        System.exit(0);
                  }
            });
            btnSalir.setFont(new Font("Tahoma", Font.PLAIN, 12));
            btnSalir.setBounds(270, 336, 70, 31);
            marco.getContentPane().add(btnSalir);
      }

      private void validarYguardar() {
            // Obtener los valores de los controles
            String nombre = txtNombre.getText().trim();
            String apellidos = txtApellidos.getText().trim();
            String direc = txtDireccion.getText().trim();
            String ciudad = txtCiudad.getText().trim();
            String provincia = txtProvincia.getText().trim();
            TarjetaTipo tipoTarjeta =
```

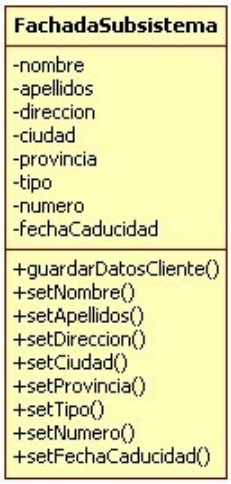
```
                  (TarjetaTipo) cmbTarjeta.getSelectedItem();
            String numTarjeta = txtNumeroTarjeta.getText().trim();
            String fechaCadTarjeta =
txtFechaCaducidad.getText().trim();

            // Proceso de validacion
            boolean formValidado = false;

            Usuario usuario =
                  new Usuario(nombre, apellidos);

            Direccion direccion =
                  new Direccion(direc, ciudad, provincia);

            Tarjeta tarjeta =
                  new Tarjeta(tipoTarjeta, numTarjeta,
fechaCadTarjeta);

            formValidado = usuario.esValido() && direccion.esValido()
                        && tarjeta.esValido();

            if (formValidado) {
                  usuario.guardarDatos();
                  direccion.guardarDatos();
                  tarjeta.guardarDatos();
            }

            String mensaje;
            if (formValidado) {
                  mensaje = " Formulario correcto: los datos se han
guardado en disco. ";
            } else {
                  mensaje = " Formulario INCORRECTO: los datos no se
han guardado. ";
            }

            lblMensaje.setText(mensaje);

      }
}
```

Notad que se ha resaltado en color amarillo el código de la clase cliente que, con tal de cumplir con los requisitos funcionales, se acopla en exceso con el subsistema. Esto es precisamente lo que queremos mejorar mediante la utilización del patrón Facade.

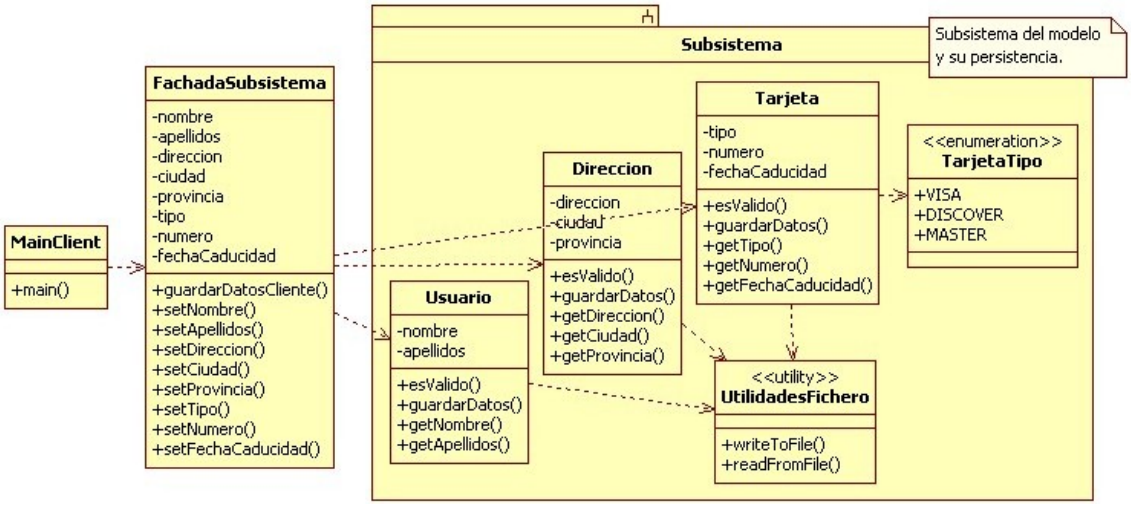
Revisando el diseño anterior: uso del patrón Facade

En esta nueva aproximación tendremos una nueva clase, FachadaSubsistema, que nos proporcionará una sencilla interfaz de acceso al subsistema del modelo (Usuario, Dirección, Tarjeta, etc.). FachadaSubsistema ofrece un método de negocio de alto

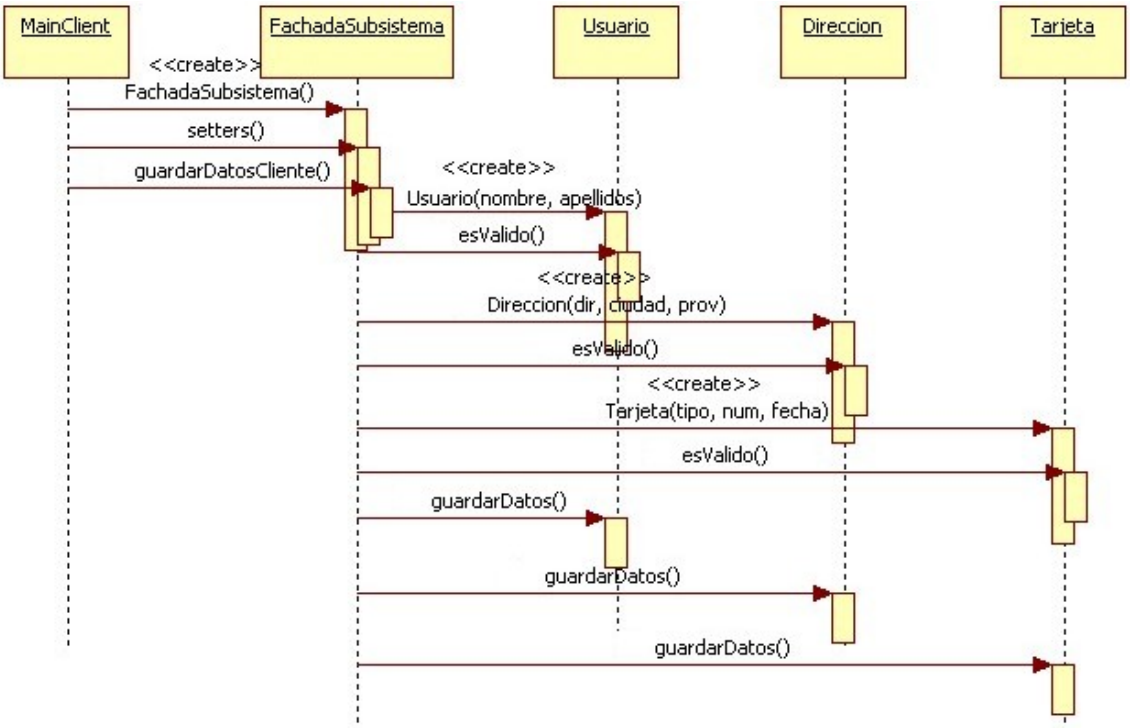
nivel llamado guardarDatosCliente() que permitirá al código cliente validar y guardar los datos del modelo.



El nuevo diseño queda expresado en la siguiente figura:



En la siguiente figura podemos observar el flujo de mensajes en el nuevo diseño. El código cliente tan sólo ha de interactuar con el nuevo componente: FachadaSubsistema. Notad también que la nueva clase cliente se llamada ahora MainClient.



FachadaSubsistema se encarga de crear los diferentes objetos del subsistema y de proceder a la validación y almacenamiento de sus datos.

FachadaSubsistema.java

```
package estructurales.facade.subsistema;

public class FachadaSubsistema {
	private String nombre;
	private String apellidos;
	private String direc;
	private String ciudad;
	private String provincia;
	private TarjetaTipo tipo;
	private String numero;
	private String fechaCaducidad;

	private static final FachadaSubsistema instance =
		new FachadaSubsistema();

	private FachadaSubsistema() {

	}

	public static FachadaSubsistema getInstance() {
		return instance;
	}

	public boolean guardarDatosCliente() {
		Direccion direccion;
		Usuario usuario;
```

```
		Tarjeta tarjeta;

		boolean datosValidos = true;
		String megError = "";

		/*
		 * El código cliente no tiene constancia de ninguna
		 * de las operaciones del subsistema.
		 */

		/*
		 * Validaciones
		 */
		usuario = new Usuario(nombre, apellidos);
		if (!usuario.esValido()) {
			datosValidos = false;
			megError = "Nombre o apellidos incorrectos";
		}

		direccion = new Direccion(direc, ciudad, provincia);
		if (!direccion.esValido()) {
			datosValidos = false;
			megError = "Direccion/Ciudad/Provincia incorrectas";
		}

		tarjeta = new Tarjeta(tipo, numero, fechaCaducidad);
		if (!tarjeta.esValido()) {
			datosValidos = false;
			megError = "Datos de la tarjeta incorrectos";
		}

		if (!datosValidos) {
			System.out.println(megError);
			return false;
		}

		/*
		 * Almacenamiento
		 */
		if (direccion.guardarDatos() &&
				usuario.guardarDatos() &&
tarjeta.guardarDatos())
		{
			return true;
		} else {
			return false;
		}

	}

	/*
	 * Setters invocados por el código cliente
	 */
	public void setNombre(String nombre) {
		this.nombre = nombre;
	}

	public void setApellidos(String apellidos) {
		this.apellidos = apellidos;
```

```
	}

	public void setDireccion(String direccion) {
		this.direc = direccion;
	}

	public void setCiudad(String ciudad) {
		this.ciudad = ciudad;
	}

	public void setProvincia(String provincia) {
		this.provincia = provincia;
	}

	public void setTipo(TarjetaTipo tipo) {
		this.tipo = tipo;
	}

	public void setNumero(String numero) {
		this.numero = numero;
	}

	public void setFechaCaducidad(String fechaCaducidad) {
		this.fechaCaducidad = fechaCaducidad;
	}
}
```

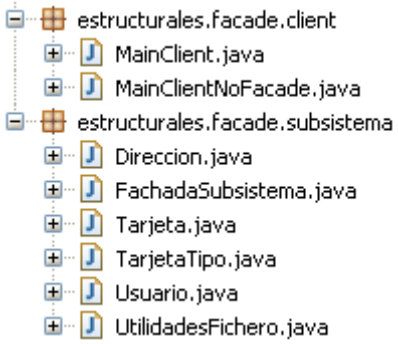
Por último, veamos el código de MainClient, que es prácticamente igual al de la clase MainClientNoFacade. De hecho, el único código que varía es el señalado en amarillo en el listado de MainClientNoFacade. Este código resaltado será sustituido por el siguiente fragmento:

MainClient.java (sólo el método validarYguardar(), el resto de MainClientNoFacade es idéntico)

```
...
private void validarYguardar() {
	// Obtener los valores de los controles
	String nombre = txtNombre.getText().trim();
	String apellidos = txtApellidos.getText().trim();
	String direc = txtDireccion.getText().trim();
	String ciudad = txtCiudad.getText().trim();
	String provincia = txtProvincia.getText().trim();
	TarjetaTipo tipoTarjeta =
		(TarjetaTipo) cmbTarjeta.getSelectedItem();
	String numTarjeta = txtNumeroTarjeta.getText().trim();
	String fechaCadTarjeta = txtFechaCaducidad.getText().trim();

	// Crear la instancia del componente facade
**********************
	FachadaSubsistema facade =
			FachadaSubsistema.getInstance();
```

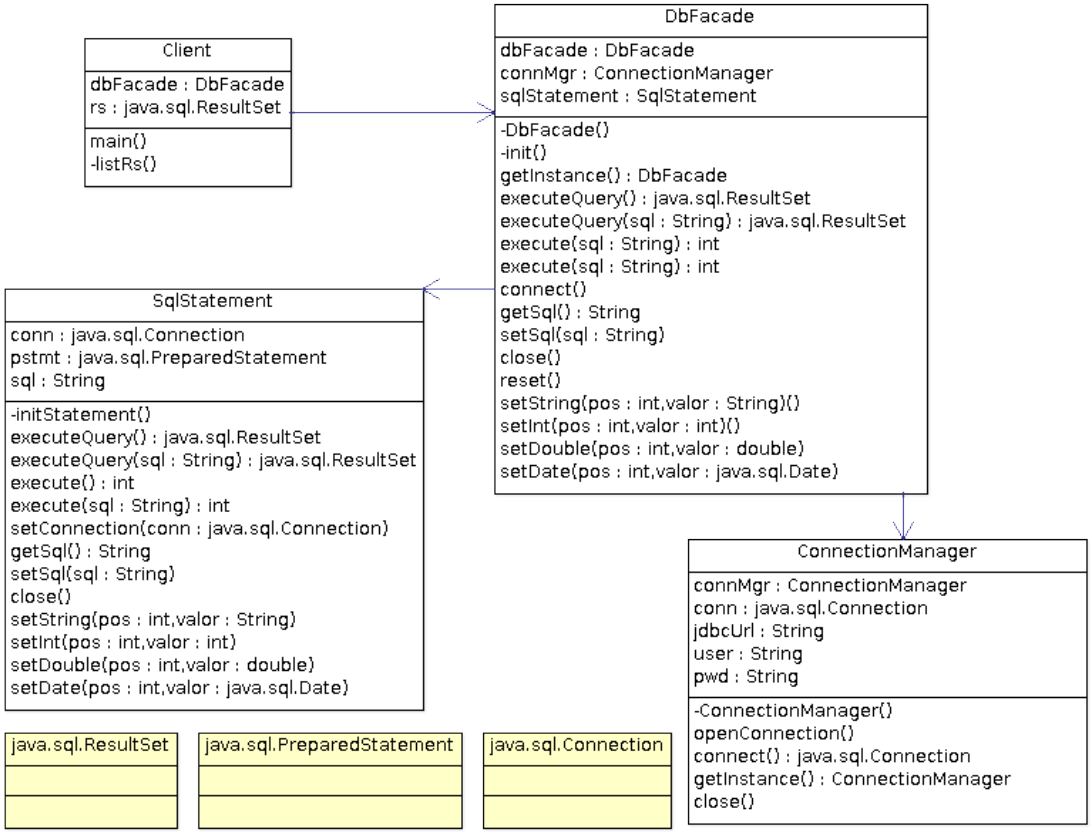
```
        // configurar la fachada
        facade.setNombre(nombre);
        facade.setApellidos(apellidos);
        facade.setDireccion(direc);
        facade.setCiudad(ciudad);
        facade.setProvincia(provincia);
        facade.setTipo(tipoTarjeta);
        facade.setNumero(numTarjeta);
        facade.setFechaCaducidad(fechaCadTarjeta);

        /*
         * El código cliente no necesita acceder al subsistema.
         * Con la siguiente simple línea se consigue validar y
         * almacenar los datos.
         */
        boolean resultado = facade.guardarDatosCliente();

        // FIN de Facade************************************************

        String mensaje;

        if (resultado) {
              mensaje = " Formulario correcto: los datos se han guardado
en disco. ";
        } else {
              mensaje = " Formulario INCORRECTO: los datos no se han
guardado. ";
        }

        lblMensaje.setText(mensaje);
}

...
```

La siguiente figura muestra las clases de la aplicación y su distribución en paquetes:



Ejemplo 2: Simplificar el acceso a datos de una clase cliente

El acceso a datos mediante JDBC implica que una clase cliente tenga que encargarse de instanciar diversos objetos del paquete java.sql o javax.sql. Esto suele ser una tarea bastante tediosa, ya que el código que realiza trabajo efectivo de acceso a datos rápidamente se ensombrece con código específico de JDBC. Aplicar aquí el patrón Facade permitirá mantener un código bastante limpio en las clases cliente. El siguiente diagrama muestra las clases que vamos a utilizar en el ejemplo:



La idea de la figura anterior, es que la clase Client sólo interactúa con la clase Facade, por lo que sólo tiene que utilizar métodos de este objeto para llevar a cabo todas las operaciones de acceso a datos. No obstante, hay que advertir que esto no es del todo cierto, ya que declara un atributo de tipo java.sql.ResulSet para poder listar los datos devueltos por una consulta.

Por tanto, tenemos un escenario en el que la clase Client implementa un código muy fácil de seguir, ya que todo el trabajo pesado lo realiza Facade en colaboración con ConnectionManager y SqlStatement.

Las clases en amarillo son las propias del paquete java.sql.

Por otro lado, tened en cuenta que se utiliza un fichero de propiedades con valores para la cadena de conexión a la base de datos. Este fichero tiene preferencia sobre los valores harcodeados en la clase ConnectionManager. El fichero debe ubicarse en el mismo paquete en el que residen las clases.

database.properties

```
user=root
pwd=root
jdbc_url=jdbc:mysql://localhost/test
```

Consideraciones sobre la tabla utilizada por el programa

El programa utiliza una tabla que se ha creado en la base de datos 'test' de MySql, que es una BD que viene por defecto en la instalación predeterminada. La estructura de la tabla es como sigue:

```
CREATE TABLE `employees` (
  `employee_id` int(11) NOT NULL,
  `firstname` varchar(50) NOT NULL,
  `surname` varchar(50) NOT NULL,
  `department_id` int(11) NOT NULL,
  PRIMARY KEY (`employee_id`))
```

Respecto al contenido de la tabla, se parte de los siguientes registros (se muestran en formato CSV):

```
employee_id,firstname,surname,department_id
1,Daniel,garcia,1
2,teresa,fernandez,1
3,jose,sanz,2
4,esteban,Lopez,2
```

```
5,Luis,Garriga,2
```

Veamos el código.

Comenzamos con ConnectionManager. Esta clase es un Singleton que se encarga de obtener una conexión con el RDBMS y de cerrarla cuando se le solicite. De alguna forma es un wrapper para la clase java.sql.Connection.

Sólo permite una conexión (siempre devuelve la conexión que obtiene por primera vez, ya que la cachea), por lo que tendríamos problemas de concurrencia en un entorno multi-usuario, puesto que al compartir la conexión sucederían incoherencias entre las acciones de los usuarios.

ConnectionManager.java

```
package ch12_Facade;


import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;
import java.net.URL;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Properties;


class ConnectionManager {


	private static ConnectionManager connMgr;
	private static Connection conn;


	private String jdbcUrl =
			"jdbc:oracle:thin:@192.168.1.202:1521:XE";
	private String user = "scott";
	private String pwd = "heyyyyy";
```

```
	// Evitar instanciacion desde el exterior
	private ConnectionManager() {

	}

	public static synchronized ConnectionManager getInstance() {
		if (connMgr == null) {
			connMgr = new ConnectionManager();
		}

		return connMgr;
	}

	public Connection connect() throws Exception {
		if (conn == null) {
			System.out.println("La conexion no esta abierta.
Iniciando conexion...");
			loadProperties();
			openConnection();
		}
		return conn;
	}

	private synchronized void openConnection() throws Exception {
		String driver = "oracle.jdbc.OracleDriver";
		Class.forName(driver).newInstance();

		System.out.println("Conectando con la base de datos...");
		conn = DriverManager.getConnection(jdbcUrl, user, pwd);
		System.out.println("Conectado!");
```

```
}

/*
 * Carga desde un fichero de propiedades el user y el pwd,
 * sobrescribiendo los valores predeterminados
 */
private void loadProperties() throws Exception {

      URL url = getClass().getResource(
                  "/ch12_Facade/database.properties");

      File file = new File(url.toURI().getPath());

      if (file.exists()) {
            InputStream is = null;
            try {
                  is = new FileInputStream(file);
                  Properties prop = new Properties();
                  prop.load(is);

                  user = prop.getProperty("user");
                  pwd = prop.getProperty("pwd");
                  jdbcUrl = prop.getProperty("jdbc_url");
            } finally {
                  if (is != null) {
                        is.close();
                  }
            }
      }
}
```

```
	public void close() throws SQLException {
		if (conn != null) {
			System.out.println("Cerrando la conexion a la base de datos...");
			conn.close();
			conn = null;
		}
	}
}
```

Ahora veamos la clase SqlStatement. Esta clase utiliza la clase java.sql.Connection y la clase java.sql.PreparedStament. De alguna forma actúa como un wrapper de la clase java.sql.PreparedStatement. La ventaja es que ofrece varios métodos de conveniencia para su clase cliente: la clase DbFacade que veremos en breve. Por otro lado, otra ventaja es que la abstrae de trabajar a nivel de JDBC.

SqlStatement.java

```
package ch12_Facade;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Date;

class SqlStatement {

	private Connection conn;
	private PreparedStatement pstmt;

	// La sentencia SQL a ejecutar
	private String sql;
```

```
	private void initStatement() throws SQLException {
		if (sql == null) {
			throw new SqlStatementException("La cadena SQL no
tiene contenido");
		}
		pstmt = conn.prepareStatement(sql);
	}

	public void setConnection(Connection conn) {
		this.conn = conn;
	}

	public String getSql() {
		return this.sql;
	}

	/*
	 * Asignar la sentencia SQL e inicializar el PreparedStatement
	 */
	public void setSql(String sql) throws SQLException {
		this.sql = sql;
		pstmt = null;
		initStatement();
	}

	/*
	 * Ejecutar la sentencia SQL estática que se recibe por
parámetro
	 */
	public ResultSet executeQuery(String sql) throws SQLException {
		setSql(sql);
```

```
            return executeQuery();
      }

      /*
       * Ejecutar una sentencia SELECT
       */
      public ResultSet executeQuery() throws SQLException {
            if (pstmt == null) {
                  throw new SqlStatementException("PreparedStatement
no inicializado");
            }
            return pstmt.executeQuery();
      }

      /*
       * Ejecutar una sentencia SQL estática de tipo
       * UPDATE, INSERT o DELETE que se recibe por parámetro
       */
      public int execute(String sql) throws SQLException {
            setSql(sql);
            return execute();
      }

      /*
       * Ejecutar una sentencia SQL de tipo
       * UPDATE, INSERT o DELETE
       */
      public int execute() throws SQLException {
            if (pstmt == null) {
                  System.out.println("PreparedStatement no
inicializado.");
            }
```

```
            int count = pstmt.executeUpdate();
            return count;
      }

      public void close() throws SQLException {
            if (pstmt != null) {
                  System.out.println("Cerrando SqlStatement...");
                  pstmt.close();
                  pstmt = null;
            }
      }

      /**
       * Los siguientes setters establecen los valores
       * apropiados en un PrepatedStatement
       */

      public void setString(int index, String value) throws
SQLException {
            pstmt.setString(index, value);
      }

      public void setInt(int index, int value) throws SQLException {
            pstmt.setInt(index, value);
      }

      public void setDouble(int index, double value) throws
SQLException {
            pstmt.setDouble(index, value);
      }
```

```
	public void setDate(int index, Date value) throws SQLException {
		pstmt.setDate(index, value);
	}
}
```

La clase anterior tiene una clase que modela una excepción:

SqlStatementException.java

```
package ch12_Facade;


import java.sql.SQLException;


class SqlStatementException extends SQLException {


	private static final long serialVersionUID = 1L;


	public SqlStatementException() {
		super();
	}


	public SqlStatementException(String message) {
		super(message);
	}
}
```

Ahora veamos DbFacade. Esta clase es un Singleton que, como se ha comentado anteriormente, evita a las clases clientes tener que utilizar objetos del paquete java.sql.

DbFacade.java

```
package ch12_Facade;


import java.sql.Connection;
import java.sql.Date;
import java.sql.ResultSet;
import java.sql.SQLException;


public class DbFacade {


	private static DbFacade dbFacade;
	private ConnectionManager connMgr;
	private SqlStatement sqlStatement;


	public static synchronized DbFacade getInstance() throws
Exception {


		if (dbFacade == null) {
			dbFacade = new DbFacade();
		}


		return dbFacade;
	}


	// Evitar instanciacion desde el exterior
	private DbFacade() throws Exception {
		connMgr = ConnectionManager.getInstance();
		init();
	}


	public void connect() throws Exception {
		Connection conn = connMgr.connect();
```

```
            if (sqlStatement == null) {
                  sqlStatement = new SqlStatement();
            }
            sqlStatement.setConnection(conn);
      }

      /*
       * Abrir una conexion y pasarla al objeto SqlStatement
       */
      private void init() throws Exception {
            Connection conn = connMgr.connect();
            sqlStatement = new SqlStatement();
            sqlStatement.setConnection(conn);
      }

      /*
       * Ejecutar la sentencia SQL estática que se recibe por
parámetro
       */
      public ResultSet executeQuery(String sql) throws SQLException {
            return sqlStatement.executeQuery(sql);
      }

      /*
       * Ejecutar una sentencia SELECT
       */
      public ResultSet executeQuery() throws SQLException {

            return sqlStatement.executeQuery();
      }

      /*
```

```
 * Ejecutar una sentencia SQL estática de tipo
 * UPDATE, INSERT o DELETE que se recibe por parámetro
 */
public int execute(String sql) throws SQLException {
	return sqlStatement.execute(sql);
}

/*
 * Ejecutar una sentencia SQL de tipo
 * UPDATE, INSERT o DELETE
 */
public int execute() throws SQLException {
	return sqlStatement.execute();
}

/*
 * Asignar la sentencia SQL al SqlStatement
 */
public void setSql(String sql) throws SQLException {
	sqlStatement.setSql(sql);
}

/*
 * Limpiar el SqlStatement
 */
public void reset() throws Exception {
	sqlStatement = null;
	init();
}

public void close() throws SQLException {
```

```
            if (sqlStatement != null) {
                  sqlStatement.close();
            }

            if (connMgr != null) {
                  connMgr.close();
            }
      }

      public void setString(int index, String value) throws
SQLException {
            sqlStatement.setString(index, value);
      }

      public void setInt(int index, int value) throws SQLException {
            sqlStatement.setInt(index, value);
      }

      public void setDouble(int index, double value) throws
SQLException {
            sqlStatement.setDouble(index, value);
      }

      public void setDate(int index, Date value) throws SQLException {
            sqlStatement.setDate(index, value);
      }

}
```

DbFacade tiene asociada una clase que modela un excepción:

DbFacadeException.java

```
package ch12_Facade;


import java.sql.SQLException;


public class DbFacadeException extends SQLException {


	private static final long serialVersionUID = 1L;


	public DbFacadeException() {
		super();
	}


	public DbFacadeException(String message) {
		super(message);
	}
}
```

Finalmente, el código de la clase cliente. Notad que el código de acceso a daots es muy directo gracias al dbFacade:

Client.java

```
package ch12_Facade;


import java.sql.ResultSet;
import java.sql.SQLException;
```

```
public class Client {

	private static ResultSet rs;
	private static DbFacade dbFacade;

	public static void main(String[] args) {
		try {

			dbFacade = DbFacade.getInstance();

			System.out.println("SELECT estatico con
executeQuery()");
			String sqlAll = "SELECT * FROM employees";

			rs = dbFacade.executeQuery(sqlAll);
			listRs();

			// *************************

			System.out.println("UPDATE parametrizado mediante
execute()" +
			"para modificar el apellido del empleado cuyo id=4");

			String sql = "UPDATE employees SET surname = ? WHERE
employee_id = ?";
			dbFacade.setSql(sql);
			dbFacade.setString(1, "Lopez");
			dbFacade.setInt(2, 4);
			dbFacade.execute();

			// *************************
```

```
                System.out.println("UPDATE estatico mediante
execute(sql)" +
                                   "para modificar el nombre del empleado
cuyo id=1");

                sql = "UPDATE employees SET firstname = 'Daniel'
WHERE employee_id = 1";
                dbFacade.execute(sql);

                // *************************

                // Listar resultados
                System.out.println("Verificar resultados de los
UPDATES");
                rs = dbFacade.executeQuery(sqlAll);
                listRs();

                // *************************

                System.out.println("Añadir nuevos empleados con
INSERT y " +
                "execute() y despues verificar resultados");
                sql = "INSERT INTO employees VALUES (?,?,?,?)";
                dbFacade.setSql(sql);

                int empleadoId = 5;
                dbFacade.setInt(1, empleadoId);

                String firstName = "Luis";
                dbFacade.setString(2, firstName);

                String lastName = "Garriga";
                dbFacade.setString(3, lastName);
```

```
                int deptId = 2;
                dbFacade.setInt(4, deptId);

                dbFacade.execute();

                // *************************

                // Listar resultados
                System.out.println("Verificar resultados del
INSERT");
                sql = "SELECT * FROM employees WHERE firstname =
'Luis'";
                rs = dbFacade.executeQuery(sql);
                listRs();

                // *************************

                System.out.println("Cerrar y abrir una conexion,
luego verificar");
                dbFacade.close();
                dbFacade.connect();
                rs = dbFacade.executeQuery(sqlAll);
                listRs();

                System.out.println("Saliendo del programa...");
                dbFacade.close();

            } catch (DbFacadeException dbe) {
                dbe.printStackTrace();
            } catch (Exception e) {
                e.printStackTrace();
```

```
		} finally {
			if (dbFacade != null) {
				try {
					dbFacade.close();
				} catch (SQLException e) {
					e.printStackTrace();
				}
			}
		}

	}

	private static void listRs() throws SQLException {
		while (rs.next()) {
			int id = rs.getInt("employee_id");
			String firstName = rs.getString("firstname");
			String lastName = rs.getString("surname");
			int deptId = rs.getInt("department_id");

			System.out.print("Fila: " + rs.getRow());
			System.out.print(", Empleado ID: " + id);
			System.out.print(", Nombre: " + firstName);
			System.out.print(", Apelllido: " + lastName);
			System.out.println(", Departamento ID: " + deptId);
		}

	}

}
```

Problemas específicos e implementación

Consideraciones al utilizar el patrón Facade:

- Facade no debe proporcionar ninguna nueva funcionalidad que no exista en el subsistema.
- Facade no debe retornar nunca al código cliente ningún tipo de datos perteneciente al subsistema. Por ejemplo, si recordamos el primer ejemplo visto, supongamos que tuviéramos un método como el siguiente en la clase FachadaSubsistema:

  ```
  public Tarjeta getTarjeta();
  ```

  Tal método estaría exponiendo el subsistema al código cliente, con lo que perderíamos parte de los beneficios de usar el patrón Facade (bajo acoplamiento, transparencia, etc).
- El objetivo de Facade es proporcionar una interfaz de alto nivel, por lo que siempre será preferible ofrecer un servicio de negocio con un alto nivel de abstracción que ofrecer una serie de tareas de bajo nivel.
- Si se necesita que varios clientes accedan a subconjuntos diferentes de la funcionalidad que proporciona el subsistema, para evitar que se acabe usando sólo una pequeña parte de la fachada, será conveniente crear varias fachadas más específicas, en lugar de una única global.

Patrones relacionados

Se puede utilizar Facade conjuntamente con Abstract Factory para proporcionar una interfaz que permita crear de manera independiente objetos de un sistema. Abstract Factory se puede utilizar como una alternativa a Facade para ocultar clases específicas a una plataforma.

El patrón Mediator es similar a Facade en tanto que también abstrae la funcionalidad de clases. No obstante, el propósito de Mediator es abstraer la comunicación entre objetos hermanos, a menudo centralizando la funcionalidad que no pertenece a ninguno de ellos. Tales objetos hermanos conocen de la existencia de Mediator y se

comunican con él, en lugar de comunicarse entre ellos. En cambio, Facade, tan sólo abstrae la interfaz de un subsistema para que sea más sencillo de utilizar, no define ninguna nueva funcionalidad y las clases del subsistema no son conscientes de su existencia.

Normalmente, sólo se necesita un objeto Facade, por lo que las fachadas suelen implementarse como Singleton’s.