9 Implementación

El modelo de implementación toma el resultado del modelo de diseño para generar el código final. Esta traducción debe ser relativamente sencilla y directa, ya que las decisiones mayores han sido tomadas durante las etapas previas. Durante el modelo de implementación se hace una adaptación al lenguaje de programación y/o la base de datos de acuerdo a la especificación del diseño y según las propiedades del lenguaje de implementación y base de datos. Aunque el diseño de objetos es bastante independeniente del lenguaje actual, todos los lenguajes tendrán sus particularidades, las cuales deberán adecuarse durante la implementación final. La elección del lenguaje influye en el diseño, pero el diseño no debe depender de los detalles del lenguaje. Si se cambia de lenguaje de programación no debe requerirse el re-diseño del sistema.

En general, no se debe comenzar prematuramente a programar, es importante primero completar el proceso de planeación del sistema final desarrollado durante el diseño. Se debe usar guías de programación existentes en la organización. Si no existen, el equipo de software deben crear sus propias guías para decidir aspectos, como formatos para la asignación de nombres a las variables, estilo de programación, métodos de documentación, y documentación en línea. Vale la pena resaltar que aunque existe cierta automatización en el proceso de generación del código final, en su gran mayoría los prgramadores hacen de manera "manual" la transición final a código fuente.

9.1 Programación en Java

En esta sección tomamos la especificación del diseño hecho en el Capítulo 8 y generamos la programación, en este caso en Java.

InterfaceUsuario

Comenzamos la implementación de la clase *InterfaceUsuario* tomando su descripción definida en la tarjeta de clase correspondiente, como se muestra en la Tabla 9.1.

Clase: InterfaceUsuario	
Descripción: Toda la interacción con el usuario se hace por medio de la interface de usuario.	
Módulo: InterfaceUsuario	
Estereotipo: Borde	
Propiedades: Concreta	
Superclase:	
Subclase:	
Atributos: Manejador, Pantalla	
Contratos	
1. Desplegar Pantalla	
desplegarPantalla(Pantalla) devuelve void *Método encargado de desplegar las pantallas enviadas como parámetros. Se delega el despliegue particular a cada pantalla.*	Pantalla (1) : PantallaPrincipal (1), PantallaServicio (1), PantallaCrearRegUsuario (1), PantallaObtenerRegUsuario (1), PantallaCrearRegTarjeta (1), PantallaObtenerRegTarjeta (1)
2. Enviar Evento	
enviarEvento(Evento) devuelve void *Método encargado de recibir eventos del sistema de ventanas. Se envía el evento recibido a los distintos manejadores.*	Manejador (1) : SubsistemaPrincipal (1), SubsistemaServicio (1), SubsistemaRegistro (1)

Tabla 9.1. Tarjeta para la clase *InterfaceUsuario* con responsabilidades, colaboraciones, jerarquías, contratos, subsistemas y protocolos identificados de los casos de uso *RegistrarUsuario*, *ValidarUsuario* y *RegistrarTarjeta*.

La implementación de la clase *InterfaceUsuario* se hará a partir de la biblioteca AWT ("java.awt") de Java y se basa en la descripción de de manejo de ventanas hecha anteriormente en el Capítulo 5. En nuestro manejo de ventanas heredamos la clase *InterfaceUsuario* de la clase *Frame* de Java para generar una sola ventana la cual mostrará diferentes pantallas dentro del marco de la misma ventana pero en diferentes momentos. Esta clase implementa los

manejadores de eventos de ventana y acciones como se describió anteriormente y se vuelve a describir a continuación.

```
public class InterfaceUsuario extends Frame
   implements WindowListener, ActionListener
```

Los atributos de la clase son de tipo *Manejador* y *Pantalla*, como se definió anteriormente. Asignamos como nombre de las variables los mismos tipos pero en minúscula y privados por ser atributos.

```
  private Manejador manejador;
  private Pantalla pantalla;
```

Los métodos a sobrescribir de estos manejadores de eventos fueron también descritos anteriormente. En el caso de eventos de ventanas se sobrescriben ("*implements*") los métodos descritos en la *interface* llamada *WindowListener*.

```
  public void windowClosed(WindowEvent event) {}
  public void windowDeiconified(WindowEvent event) {}
  public void windowIconified(WindowEvent event) {}
  public void windowActivated(WindowEvent event) {}
  public void windowDeactivated(WindowEvent event) {}
  public void windowOpened(WindowEvent event) {}
  public void windowClosing(WindowEvent event) {
    System.exit(0);
  }
```

En el caso de eventos relacionados con acciones (botones) se debe sobrescribir un solo método de la *interface* llamada *actionListener*, el método *actionPerformed*. Este método es el punto de entrada de los eventos administrados por el manejador de ventanas del sistema y provenientes del *Usuario*. Recordemos que la clase *InterfaceUsuario* define un contrato "2" llamado *enviarEvento*, el cual debe recibir los eventos relacionados con los botones para luego enviárselos a los distintos manejadores. Este método *enviarEvento* corresponde a *actionPerformed* y debe rescribirse con dicho nombre, como se muestra a continuación. El parámetro, definido originalmente como *Evento* corresponde ahora a *ActionEvent* de Java.

```
   // Contrato 2: Enviar Evento (ActionListener)
  public void actionPerformed(ActionEvent event)
```

Aprovechando que ya estamos definiendo el método correspondiente al contrato "2", veamos como debe implementarse. El método, como se especificó anteriormente, tiene como responsabilidad renviar el evento a los diversos manejadores. Aunque pudiéramos renviar el mismo tipo de evento *ActionEvent* a los menjadores, esteríamos algo limitados en la extensiblidad del sistema, ya este tipo pudiera cambiar en un futuro dependiendo de las bibliotecas particular utilizadas. En su lugar, pudiéramos definir nuestro propio tipo de evento para comunicación interna del sistema, o en el caso de eventos sencillos, como los que manejaremos aquí, simplemente enviar una *String* correspondiente al nombre del botón presionado. El nombre del botón lo podemos obtener mediante la llamada *event.getActionCommand()*. La llamada a los manejadores será a través del contrato "1" de estos, "Manejar Evento", específicamente la responsabilidad *manejarEvento* la cual fue definida para recibir un parámetro de tipo *String*. Este método es sobrescrito por todos los manejadores, donde la llamada se hace a partir de la referencia genérica de *manejador*. La implementación del método se muestra a continuación. La llamada se hace dentro de un "if-else" para asegurar que no exista una referencia nula.

```
    System.out.println("Action: "+event.getActionCommand());
    if (manejador != null)
      manejador.manejarEvento(event.getActionCommand());
    else
       System.out.println("Manejador nulo");
```

Ya definido el manejo de evento, tanto de ventana como de acciones (contrato "2"), continuamos con el despliegue de pantallas, correspondiente al contrato "1" "Desplegar Pantallas", el cual tiene un solo método definido, *desplegarPantalla*, el cual tiene como parámetro el tipo general Pantalla, como se muestra a continuación.

```
  // Contrato 1: Desplegar Pantalla
  public void desplegarPantalla(Pantalla p)
```

De manera similar al contrato "1" de "Manejar Evento", este contrato tiene como responsabilidad solicitar a las diversas pantallas que se desplieguen. Sin embargo, antes de desplegar la siguiente pantalla, debemos borrar la actual. Esto se hace mediante un nuevo método que definiremos en la clase *Pantalla*, la cual se encargará de borrar los elementos particulares de las pantalla. Aunque pudiéramos hacerlo a nivel general de la *InterfaceUsuario*, existen elementos conocidos por cada pantalla, como los botones, que en el case de la biblioteca AWT sería importante deshabilitar antes de proseguir con la añadición de nuevos botones a una misma ventana. El nuevo método lo

llamaremos *borrarPantalla* y lo llamaremos a partir de la referencia genérica de *pantalla*, siempre revisando que la referencia no sea nula, como se muestra a continuación.

```
    if (pantalla != null)
        pantalla.borrarPantalla();
```

A continuación, estamos listo para desplegar nuestra nueva pantalla. Como paso preliminar asignamos el valor de la referencia de "p" enviada como parámetro a la referencia local *pantalla*.

```
    if (p != null)
        pantalla = p;
```

Hecho esto estamos listos para desplegar las nuevas pantallas mediante la solicitud al contrato "1" de las diversas pantallas, correspondiente a la responsabilidad *desplegarPantalla*. Como hicimos anteriomente checamos que el valor de *pantalla* no sea nulo y hacemos la llamada de despliegue.

```
    if (pantalla != null)
       pantalla.desplegarPantalla();
```

Finalmente, lo único que queda es pedir al manejador de ventanas que muestre la nueva pantalla, algo que sea hace con la llamada de *show* definida en la clase *Frame*, superclase de *InterfaceUsuario*.

```
    show();
```

Finalmente, debemos definir el constructor para la clase. La definición es bastante similar a la descrita en el Capítulo 5, la diferencia principal radica en que el constructor es llamado por la clase *ManejadorPrincipal* como parte del proceso de inicialización, por lo cual se debe agregar el parámetro correspondiente en el constructor, como se muestra a continuación.

```
  public InterfaceUsuario(Manejador m)
```

El cuerpo del cosntructor es similar a como se describión en el Capítulo 5, donde simplemente asignamos el parámetro "m" a la referencia local *manejador*.

```
    setSize(800,600);
    setBackground(Color.lightGray);
    addWindowListener(this);
    manejador = m;
```

Como podemos observar, la definición de InterfaceUsuario se basa en el diseño del Capítulo 5, adaptando los contratos definidos durante el diseño.

La clase *Pantalla* tomando su descripción definida en la tarjeta de clase correspondiente, como se muestra en la Tabla 9.2.

Clase: Pantalla	
Descripción: Pantalla heredada por las demás clases de tipo pantalla.	
Módulo: InterfaceUsuario	
Estereotipo: Borde	
Propiedades: Abstracta	
Superclase:	
Subclase: PantallaPrincipal, PantallaServicio, PantallaRegUsuario, PantallaRegTarjeta	
Atributos: InterfaceUsuario, Manejador	
Contratos	
1. Desplegar Pantalla	
desplegarPantalla() devuelve void *Método encargado de desplegar la pantalla actual.*	

Tabla 9.2. Tarjeta para la superclase *Pantalla*.

La clase *Pantalla* se define como una clase abstracta. Dado que la pantalla existe dentro de un marco (*InterfaceUsuario* hereda de *Frame*), no es necesario agregar ninguna herencia a esta clase.

```
public abstract class Pantalla
```

Los atributos de la clase son de tipo *InterfaceUsuario* y *Manejador*, como se definió anteriormente. Asignamos como nombre de las variables los mismos tipos pero en minúscula y privados ya que minimizaremos su efecto a la superclase *Pantalla* y no a cada una de ellas por separado. Si esto no funciona, cambiaríamos la visibilidad a *protected*.

```
  private InterfaceUsuario interfaceUsuario;
  private Manejador manejador;
```

Como parte de la implementación agregamos también el diseño interno de las ventanas siguiendo el ejemplo desarrollado en el Capítulo 5. Definimos cuatro vectores correspondientes a todos los elementos que queremos administrar dentro de cada pantalla: paneles, botones, textos y etiquetas. Para cada uno de ellos definimos una variable temporal, las cuales utilizaremos como referencias temporales a los objetos instanciados.

```
  protected Vector paneles,botones,textos,etiquetas;
  protected Panel panel;
  protected Button boton;
  protected TextField texto;
  protected Label etiqueta;
```

El constructor de la pantalla debe recibir las referencias de la InterfaceUsuario y del manejador que acaba de instanciar la pantalla, y debe permitir inicializar y crear los elementos internos de la ventana, algo que hacemos con los métodos `inicializarPantalla` y `crearPantalla`, respectivamente.

```
  public Pantalla(InterfaceUsuario ui,Manejador m) {
    interfaceUsuario = ui;
    manejador = m;
    inicializarPantalla();
    crearPantalla();
  }
```

El método `inicializarPantalla` y se encarga de inicializar los vectores como se muestra continuación. Definimos tanto los métodos locales a las pantallas como aquellos que son llamados por la clase *InterfaceUsuario* como protegidos, como se mostró anteriormente en el Capítulo 5.

```
  protected void inicializarPantalla() {
    paneles = new Vector();
    botones = new Vector();
    textos = new Vector();
    etiquetas = new Vector();
  }
```

El método `crearPantalla` se define como abstracto y copmo protegido.

```
  protected abstract void crearPantalla();
```

Otro método que ha sido mencionado anteriormente y que se define también como protegido es `borrarPantalla`, el cual está encargado de borrar los elementos de una pantalla en el momento de desplegar una nueva.

```
  protected void borrarPantalla() {
    interfaceUsuario.removeAll();
    int bs = botones.size();
    for (int i = 0; i < bs; i++)
    if ((boton = (Button)botones.elementAt(i)) != null)
      boton.removeActionListener(interfaceUsuario);
  }
```

Otros dos métodos aún no mencionados que aprovecharemos dado que muchas clases agregan botones de “Salir” y de “Servicios” son los siguientes. Definimos dos ya que hay pantallas que sólo requieren “Salir”, mientras que otras requieren ambas.

```
  protected void agregarBotonesSalir(Panel panel){
    boton = new Button ("Salir");
    panel.add(boton);
    botones.addElement(boton);
    paneles.addElement(panel);
  }
  protected void agregarBotonesServiciosSalir(Panel panel){

    boton = new Button ("Servicios");
    botones.addElement(boton);
    panel.add(boton);

    agregarBotonesSalir(panel);
  }
```

El contrato "1", "Desplegar Pantalla", consiste de la responsabilidad *desplegarPantalla* la cual también fue definida y explicada en el Capítulo 5. La volvemos a mostrar, esta vez como parte del contrato. También la definimos como protegida ya que es llamada por la *InterfaceUsuario*, la cual está definida como parte del mismo paquete.

```
  // Contrato 1: Desplegar Pantalla
  protected void desplegarPantalla() {
    System.out.println("Desplegando: "+ this);
    int ps = paneles.size();
    interfaceUsuario.setLayout(new GridLayout(ps,1));
    for (int i = 0; i < ps; i++)
      interfaceUsuario.add((Panel)paneles.elementAt(i));
    int bs = botones.size();
    for (int i = 0; i < bs; i++)
      if ((boton = (Button)botones.elementAt(i)) != null)
        boton.addActionListener(interfaceUsuario);
  }
```

Principal

A continuación se describe la clase `Manejador`, como se muestra en la Tabla 9.3.

Clase: Manejador	
Descripción: Pantalla heredada por las demás clases de tipo pantalla.	
Módulo: Principal	
Estereotipo: Control	
Propiedades: Abstracta	
Superclase:	
Subclase: ManejadorPrincipal, ManejadorServicio, ManejadorRegistroUsuario, ManejadorRegistroTarjeta	
Atributos: InterfaceUsuario, Pantalla, ManejadorServicio, Manejador	
Contratos	
1. Manejar Evento	
manejarEvento(Evento) devuelve void *Método encargado de recibir eventos del sistema de ventanas a través de la InterfaceUsuario.*	
Responsablidades Privadas	
desplegarPantalla() devuelve void *Método encargado de desplegar las pantallas administradas por los manejadores. Se solicita al SubsistemaInterfaceUsuario que las despliegue.*	SubsistemaInterfaceUsuario (1)
manejarEventoOfrecerServicio() devuelve void *Método encargado de solicitar al SubsistemaServicio que ofrezca los servicios correspondientes.*	SubsistemaServicio (2)
manejarEventoSalir() devuelve void *Método encargado de salir del sistema.*	

Tabla 9.3. Tarjeta para la clase *Manejador* con responsabilidades, colaboraciones, jerarquías, contratos, subsistemas y protocolos identificadas de los diversos manejadores para los casos de uso *RegistrarUsuario*, *ValidarUsuario* y *RegistrarTarjeta*.

La clase *Manejador* es la superclase de todos los manejadores y se define como abstracta.

```
public abstract class Manejador
```

Los atributos principales para la clase son de tipo `InterfaceUsuario`, `Pantalla` y `ManejadorServicio`.

```
  protected InterfaceUsuario interfaceUsuario;
  protected Pantalla pantalla;
  protected ManejadorServicio ms;
```

El constructor de la clase `Manejador` guarda la referencia a la `interfaceUsuario` y a `mPadre`, donde este último se refiere a la clase manejador encargada de instanciar a la actual.

```
  public Manejador(Manejador m,InterfaceUsuario ui) {
      interfaceUsuario = ui;
      mPadre = m;
  }
```

El contrato "1", "Manejar Evento", define la responsabilidad `manejarEvento`. Aunque se especificó originalmente un parámetro de tipo `Evento` en el protocolo, lo modificaremos a un `String` según explicamos anteriormente en la implementación del método `actionPerformed` de la clase `IntefaceUsuario`.

```
  // Contrato 1: Manejar Evento
  public abstract void manejarEvento(String str);
```

La responsabilidad `desplegarPantalla` es un buen ejemplo de una responsabilidad cuya definición de su protocolo dejamos pendiente por ser privada. Dado que esta responsabilidad será llamada por los diversos manejadores para desplegar alguna pantalla, podemos modificar su protocolo para ser protegida su visibilidad, de manera que pueda ser llamada por las subclase, además de agregar un parámetro de tipo `Pantalla` correspondiente a la pantalla a ser desplegada. El cuerpo del método debe solictar a la clase `InterfaceUsuario` proceder con el nuevo despliegue. Adicionalmente se debe actualizar la referencia local a cual `pantalla` se está desplegando y también actualizar la referencia del `manejador` registrado con la `InterfaceUsuario`.

```
  protected void desplegarPantalla(Pantalla p) {
      pantalla = p;
      if (pantalla != null) {
        interfaceUsuario.setManejador(this);
        interfaceUsuario.desplegarPantalla(p);
      }
      else
        System.out.print("Pantalla Nula");
  }
```

Debemos también agregar un método `manejarEventoOfrecerServicio` el cual solicita a la clase `ManejadorServicio` que ejecute su contrato de "Ofrecer Servicio".

```
  protected void manejarEventoOfrecerServicio () {
     if (ms != null)
            ms.ofrecerServicio();
     else
            System.out.println("No se ha inicializado el ManejadorServicios");
  }
```

Nótese que es necesario instanciar la clase `ManejadorServicio` a través de su referencia `ms`. Para ello agregamos las siguientes dos líneas en el constructor de todos los manejadores de manera que siempre obtengan la referencia del `ManejadorServicio` del manejador padre. Es necesario que algún manejador haga la isntanciación apropiada de este objeto, algo que haremos en el constructor del `ManejadorPrincipal`.

```
              if (mPadre != null)
                 ms = mPadre.getManejadorServicio();
```

El método `manejarEventoSalir` lo definimos de la siguiente manera.

```
  protected void manejarEventoSalir() {
     System.exit(0);
  }
```

Agregamos un método adicional para llamar a los dos métodos anteriores, `manejarEventoOfrecerServicio` y `manejarEventoSalir`, correspondiente a los botones presionados en las diferentes pantallas. Este método, `manejarEventosAdicionales`, puede ser llamado por los diversos métodos manejarEvento de cada manejador sin tener que duplicar el código de manera local.

```
  protected void manejarEventosAdicionales(String str) {
     if (str.equals("Servicios"))
        manejarEventoOfrecerServicio();
     else if (str.equals("Salir"))
        manejarEventoSalir();
     else
        System.out.println("Error en pantalla: "+this+", Evento: "+str);
  }
```

La clase *ManejadorPrincipal* se describe en la Tabla 9.4.

Clase: ManejadorPrincipal	
Descripción: El manejador principal es el encargado de desplegar la pantalla principal de interacción con el usuario, y luego delegar las diferentes funciones a los manejadores especializados apropiados.	
Módulo: Principal	
Estereotipo: Control	
Propiedades: Concreta	
Superclase: Manejador	
Subclase:	
Atributos: PantallaPrincipal, ManejadorServicio, ManejadorRegistroUsuario	
Contratos	
1. Manejar Evento	
manejarEvento(Evento) devuelve void *Método sobrescrito de la clase Manejador, encargado de recibir eventos del sistema de ventanas a través de la InterfaceUsuario.*	
Responsablidades Privadas	
manejarEventoRegistrar() devuelve void *Método encargado de solicitar al SubsistemaRegistro que de servicio al contrato de "Registrar Usuario".*	SubsistemaRegistro (2)
manejarEventoValidar() devuelve void *Método encargado de solicitar al SubsistemaServicio que de servicio al contrato de "Ofrecer Servicio".*	SubsistemaRegistro (2)

Tabla 9.4. Tarjeta para la clase *ManejadorPrincipal* con responsabilidades, colaboraciones, jerarquías, contratos, subsistemas y protocolos identificadas de los casos de uso *RegistrarUsuario*, *ValidarUsuario* y *RegistrarTarjeta*.

La clase `ManejadorPrincipal` es una subclase de `Manejador`, por lo cual debe definir la extensión correspondiente.

```
public class ManejadorPrincipal extends Manejador
```

Los atributos de la clase `ManejadorPrincipal` son `PantallaPrincipal`, `ManejadorServicio` y `ManejadorRegistroRegistro`. Como veremos más adelante, los diversos manejadores hacen referencia a estos últimos dos manejadores por lo cual aprovechamos para definirlos en la superclase `Manejador` en lugar de estar redefiniéndolos en cada subclase especializada. De tal manera aprovechamos el reuso de código a través de la herencia.

```
  private Pantalla pantallaPrincipal;
  private ManejadorServicio ms;
  private ManejadorRegistroUsuario mru;
```

Dado que el `ManejadorPrincipal` es el encargado de inicializar la aplicación, definimos el método estático `main` como parte de la definicón de esta clase.

```
   public static void main(String[] args) {
        ManejadorPrincipal m = new ManejadorPrincipal();
   }
```

El constructor de la clase debe instanciar a la `InterfaceUsuario` y también a los manejadores con los cuales se comunicara, el `ManejadorServicio` y el `ManejadorRegistroUsuario`. Al objeto referido por `interfaceUsuario`, definido en la clase `Manejador`, le pasamos una referencia local del `ManejadorPrincipal`, mediante un "`this`". Al objeto referido por `ms`, correspondiente a un objeto de tipo `ManejadorServicio` y también definido en la superclase `Manejador`, se le pasa la referencia local "`this`" y

también una referencia `interfaceUsuario`, para que pueda luego comunicarse al momento de desplegar pantallas. Hacemos algo similar con el `ManejadorRegistroUsuario` a través de la referencia, `mru`. Esta referencia, que aún no ha sido declarada, pudiera agregarse a esta clase o directamente a la superclase `Manejador`. Dado que también el `ManejadorSevicio` necesitará una referencia al `ManejadorRegistroUsuario`, aprovechamos para agregarla a la superclase `Manejador`. Como paso adicional, enviamos esta referencia a la clase `ManejadorServicio` mediante un nuevo método público `setManejadorRegistroUsuario` que deberá ser definido, en este caso lo haremos a nivel de la superclase `Manejador`. Finalmente, la lógica de la inicialización de la aplicación continúa con el despliegue de la `PantallaPrincipal` mediante el método local `desplegarPantallaPrincipal`.

```
public ManejadorPrincipal() {
       interfaceUsuario = new InterfaceUsuario(this);
       ms = new ManejadorServicio(this,interfaceUsuario);
       mru = new ManejadorRegistroUsuario(this,interfaceUsuario);
       ms.setManejadorRegistroUsuario(mru);
       desplegarPantallaPrincipal();
}
```

El método local, añadido durante la implementación, hace una llamada al método `desplegarPantalla` definido en la superclase `Manejador`, agregando como parámetro la refencia a la nueva pantalla recién instanciada. En general, estas instanciaciones pueden hacerse de manera dinámica, como en este ejemplo, o durante la incialización del manejador. Cual enfoque tomar depende de las consideraciones de rendimiento en relación a uso de memoria. Instanciaciones dinámicas son hechas únicamente cuando se necesitan, a diferencia de las hechas durante una inicialización. Sin embargo, esto tiene el costo de tiempo adicional durante la ejecución del programa, a diferencia de lo que se hace inicialmente de manera única, que es notado sólo al principio, pero no durante el transcurso del programa. Nótese de manera adicional que la instanciación de la clase `PantallaPrincipal` requiere de un parámetro interfaceUsuario y otro de tipo `Manejador`, a través del "`this`".

```
private void desplegarPantallaPrincipal() {
    if (pantallaPrincipal == null)
          pantallaPrincipal = new PantallaPrincipal(interfaceUsuario,this);
    desplegarPantalla(new PantallaPrincipal(interfaceUsuario,this));
}
```

Pasamos al contrato "1" de la clase `ManejadorPrincipal`, "Manejar Evento", el cual sobrescribe al contrato definido en la superclase `Manejador`. Este contrato, definido mediante el método `manejarEvento` requiere la misma firma (protocolo) que el definido en la superclase, en otras palabras, debe ser de tipo `String`. Aquí se hace el manejo de las diferentes opciones presentes en la `PantallaPrincipal`, básicamente, los botones "Registrarse por Primera Vez", "OK" y "Salir". Cada uno de estos botones aparece como una opción adicional dentro del "if-else". En el caso de "Registrarse por Primera Vez" se llama al método `manejarEventoRegistrar`, en el caso "OK" se llama al método `manejarEventoValidar`, y en el caso de "Salir" se hace el manejo a través de la superclase `Manejador` mediante la llamada `manejarEventosAdicionales` con el nombre del `String` correspondiente al evento.

```
// Contrato 1: Manejar Evento
public void manejarEvento(String str) {
     if (str.equals("Registrarse por Primera Vez"))
         manejarEventoRegistrar();
     else if (str.equals("OK")) {
         manejarEventoValidar();
     }
     else
         manejarEventosAdicionales(str);
}
```

El método `manejarEventoRegistrar` hace una llamada a la responsabilidad con nombre similar dentro de la clase `ManejadorRegistroUsuario`, representada por la referencia `mru` y correspondiente al contrato "Registrar Usuario" de esta última.

```
  private void manejarEventoRegistrar() {
        if (mru != null)
              mru.crearRegistroUsuario();
        else
              System.out.println("No se ha inicializado el ManejadorRegistroUsuario");
  }
```

De manera similar, el método `manejarEventoValidar` hace una llamada a la responsabilidad con nombre similar dentro de la clase `ManejadorRegistroUsuario`, representada por la referencia `mru` y correspondiente al cotnrato "Registrar Usuario" de esta última. A diferencia de la creación de un nuevo, registro, la validación requiere del "login" y "contraseña", ambos obtenido de la `PantallaPrincipal` mediante la llamada `leerTexto`, algo que será explicado más adelante. Una vez validado el usuario ("if" interno), se continúa con `ofrecerServicio` o con el despliegue de alguna pantalla de mensaje de error. En nuestro caso, volvemos a desplegar la misma pantalla para mantener limitado el código de nuestro ejemplo.

```
  private void manejarEventoValidar() {
        String log = pantalla.leerTexto("login");
        String pass = pantalla.leerTexto("password");
        if (mru != null) {
              if (mru.validarRegistroUsuario(log,pass) == true)
                manejarEventoOfrecerServicio();
              else
                desplegarPantalla(pantalla);
        }
        else
              System.out.println("No se ha inicializado el ManejadorRegistroUsuario");
  }
```

La descripción de la clase `PantallaPrincipal` es bastante limitada ya que no incluye responsabilidades. Sin embargo, vale la pena describirla como ejemplo a seguir para las demás pantallas. La tarjeta de clase se muestra en la Tabla 9.5.

Clase: PantallaPrincipal	
Descripción: Pantalla principal (*P*-1).	
Módulo: Principal	
Estereotipo: Borde	
Propiedades: Concreta	
Superclase: Pantalla	
Subclase:	
Atributos:	

Tabla 9.5. Tarjeta para la clase *PantallaPrincipal* con responsabilidades, colaboraciones, jerarquías y contratos identificadas de los casos de uso *RegistrarUsuario*, *ValidarUsuario* y *RegistrarTarjeta*.

La clase `PantallaPrincipal` hereda de la superclase `Pantalla`.

```
public class PantallaPrincipal extends Pantalla
```

El constructor de la clase pasa los parámetros de instanciación, `ui` y `m`, a la superclase, mediante el método `super`. Los parámetros corresponden a las referencias que debe mantener toda pantalla, una a la `InterfaceUsuario` y otra al `Manejador` que administra la pantalla.

```
  public PantallaPrincipal (InterfaceUsuario ui,Manejador m) {
    super(ui,m);
  }
```

Aunque pudiéramos haber creado los elementos internos de la pantalla dentro del constructor, preferimos hacerlo en un método adicional, `crearPantalla`, lo cual nos da más flexibilidad en el código, ya que podemos generar los eventos en el momento que deseemos y no necesariamente durante la instanciación del objeto. Sólo mostramos parte del cogió interno del método, ya que el detalle fue explicado en el Capítulo 5.

```
protected void crearPantalla() {
  panel = new Panel();
  panel.setLayout(new GridLayout(3,1));
  panel.add(new Label("SISTEMA DE RESERVACIONES DE VUELO", Label.CENTER));
  panel.add(new Label("Pantalla Principal (P-1)", Label.CENTER));
  paneles.addElement(panel);
     ...
  panel = new Panel();
  panel.add(new Label("Login:", Label.LEFT));
  texto = new TextField(20);
  texto.setName("login");
  textos.addElement(texto);
  panel.add(texto);
  paneles.addElement(panel);

  panel = new Panel();
  panel.add(new Label("Password:"));
  texto = new TextField(20);
  texto.setName("password");
  texto.setEchoChar('#');
  textos.addElement(texto);
  panel.add(texto);
  paneles.addElement(panel);
     ...
}
```

Vale la pena resaltar ciertos cambios en el manejo de lementos gráficos que aún no han sido comentados. Estos cambios tienen que ver principalmente con el manejo de los campos de texto, ya que de allí obtendremos información insertada por el usuario. Para lograr un manejo generalizado de esta información, agregamos un nombre particular a cada campo de texto. En el caso de "login", sería

```
texto.setName("login");
```

Una vez creado el texto, lo agregamos a la lista de textos, de manera similar a los paneles y botones:

```
textos.addElement(texto);
```

En el caso de "password", agregamos la opción de cambiar el carácter de despliegue a un "#", mediante:

```
texto.setEchoChar('#');
```

Entre lso cambios anteriores, el más importante para nuestro diseño es el hecho que se agregó un nombre para cada campo de texto. Este nombre luego lo utilizaremos para identificar campos en las pantallas y así obtener de manera generalizada su información o escribir en ella, algo que veremos más adelante.

Dominio

Aunque pudiéramos definir los diversos atributos de cada clase entidad en las clases correspondientes, haremos un pequeño rediseño mediante generalizando un poco la especificación de estos atributos a partir de la clase `Datos`. Para ello definiremos dos colecciones de objetos, una correspondiente al nombre del atributo y otro al valor que guarda. En el caso de los nombres, la colección debe guardar tipos `String`, mientras que en el caso de los valores, pueden corresponder a cualquier tipo de objeto. Para nuestro ejemplo utilizaremos valores también de tipo cadenas, aunque esto en general restringe un poco el manejo de los datos. En todo caso, es sencillo extender este diseño a los demás tipos, como `Integer`, etc. Por lo tanto la descripción de la clase `Datos` se muestra en la Tabla 9.6.

Clase: Datos	
Descripción: Superclase para todas las clases entidad.	
Módulo: Dominio	
Estereotipo: Entidad	
Propiedades: Abstracta	
Superclase:	
Subclase: RegistroUsuario, RegistroTarjeta	
Atributos: Colección nombres, Colección valores	

Tabla 9.6 Superclase entidad *Datos* para los casos de uso *RegistrarUsuario*, *ValidarUsuario* y *RegistrarTarjeta*.

La definición de la clase es la siguiente.

```
public class Datos
```

Podemos escoger cualquier tipo de colección como base para nuestros atributos. Para no complicarnos demasiado escogeremos el tipo `Vector` de Java, y dado que son dos los vectores que necesitaremos, pués haremos un arreglo de dos vectores. Esto se declara de la siguiente manera.

```
  protected Vector campos[];
```

El constructor de la clase Datos debe inicializar el arreglo de dos vectores junto con los propios vectores.

```
  public Datos() {
    campos = new Vector[2];
    for (int i = 0; i < 2; i++)
      campos[i] = new Vector();
  }
```

Luego necesitamos agregar un grupo de métodos para manipular la información. El primero será agregarAtributo, el cual sera llamado por las subclases cuando deseen incluir atributos en su definición. Para ello, se pasa el nombre del atributo como primer parámetro, el cual se agrega al primer vector, y un segundo parámetro correspondiente a su valor, el cual se agrega al segundo vector. En este caso pasamos un valor de inicialziación vacío de tipo `String`. En el caso de otros tipos valores, se sobrecargaría el método de manera correspondiente. También sería necesario agregar métodos adicionales para obtención o cambios en los valores.

```
  protected void agregarAtributo(String nombre, String valor) {
    campos[0].addElement(nombre);
    campos[1].addElement(valor);
  }
```

Para leer los nombres o valores guardados en los vectores, definimos los siguiente dos métodos, `leerNombre` y `leerValor`, de manera correspondiente. Dado que los vectores se accesan como arreglos, el parámetro que se pasa es el índice del atributo.

```
  public String leerNombre(int i) {
    return (String) campos[0].elementAt(i);
  }
  public String leerValor(int i) {
    return (String) campos[1].elementAt(i);
  }
```

De manera análoga, definimos un método `escribirValor` para escribir valores a los atributos.

```
  public void escribirValor(int i, String str) {
    campos[1].setElementAt(str,i);
  }
```

Finalmente, podemos definir un método que simplemente obtenga el número de atributos definidos, lo cual corresponde al tamaño del vector.

```
  public int numeroAtributos() {
    return campos[0].size();
  }
```

Registro

En el módulo de *Registro*, compuesto de los módulos *Usuario*, *Tarjeta* e *InterfaceBD*, sólo se modifican las clases de control pero no las de interface o entidad, como se verá a continuación.

Usuario

En la Tabla 9.7 se muestra la clase `ManejadoRegistroUsuario`.

Clase: ManejadorRegistroUsuario	
Descripción: El manejador de registro de usuario se encarga de todo lo relacionado con registro del usuario para poder utilizar el sistema.	
Módulo: Registro.Usuario	
Estereotipo: Control	
Propiedades: Concreta	
Superclase: Manejador	
Subclase:	
Atributos: PantallaCrearRegUsuario, PantallaObtenerRegUsuario, RegistroUsuario, ManejadorRegistrTarjeta, InterfaceBaseDatosRegistro	
Contratos	
1. Manejar Evento	

manejarEvento(Evento) devuelve void *Método sobrescrito de la clase Manejador, encargado de recibir eventos del sistema de ventanas a través de la InterfaceUsuario.*	
2. Registrar Usuario	
crearRegistroUsuario() devuelve void *Método encargado de solicitar a la InterfaceBaseDatosRegistro la creación de un nuevo RegistroUsuario a través del contrato de "Registrar Usuario"*	
validarRegistroUsuario(String,String) devuelve Boolean *Método encargado de solicitar a la InterfaceBaseDatosRegistro la validación de un usuario a través del contrato de "Registrar Usuario"*	InterfaceBaseDatosRegistro (1)
obtenerRegistroUsuario() devuelve void *Método encargado de solicitar a la InterfaceBaseDatosRegistro la obtención de un RegistroUsuario a través del contrato de "Registrar Usuario"*	
Responsabilidades Privadas	
manejarEventoRegistrar() devuelve void *Método encargado de solicitar a la InterfaceBaseDatosRegistro la creación de un nuevo RegistroUsuario a través del contrato de "Registrar Usuario"*	InterfaceBaseDatosRegistro (1)
manejarEventoActualizar() devuelve void *Método encargado de solicitar a la InterfaceBaseDatosRegistro la actualización de un RegistroUsuario a través del contrato de "Registrar Usuario"*	InterfaceBaseDatosRegistro (1)
manejarEventoEliminar() devuelve void *Método encargado de solicitar a la InterfaceBaseDatosRegistro la eliminación de un RegistroUsuario a través del contrato de "Registrar Usuario"*	InterfaceBaseDatosRegistro (1)
manejarEventoRegistrarTarjeta() devuelve void *Método encargado de solicitar a la ManejadorRegistroTarjeta que procese el contrato "Registrar Tarjeta"*	ManejadorRegistroTarjeta (2)

Tabla 9.7. Tarjeta para la clase *ManejadoRegistroUsuario* con responsabilidades, colaboraciones, jerarquías, contratos, subsistemas y protocolos identificadas de los casos de uso *RegistrarUsuario* y *ValidarUsuario*.

La clase `ManejadoRegistroUsuario` hereda de la superclase `Manejador` y se define de la siguiente manera.

```
public class ManejadorRegistroUsuario extends Manejador
```

Los atributos que definiremos para la clase son los siguientes.

```
  private Pantalla pantallaCrearRegUsuario;
  private Pantalla pantallaObtenerRegUsuario;
  private RegistroUsuario registroUsuario;
  private ManejadorRegistroTarjeta mrt;
  private InterfaceBaseDatosRegistro interfaceRegistro;
```

El constructor se encarga de instanciar los diversos objetos. Las dos pantallas se instanciarán de manera dinámica, mal igual que el `ManejadorRegistroTarjeta`.

```
  public ManejadorRegistroUsuario(Manejador m,InterfaceUsuario ui) {
      super(m,ui);
      registroUsuario = new RegistroUsuario();
      interfaceRegistro = new InterfaceBaseDatosRegistro();
  }
```

El contrato "1" Manejar Evento" debe contemplar los diversos botones que aprecen en las pantallas administradas por este manejador. En el caso de "Registrar", se llama al método `manejarEventoRegistrar`, "Actualizar" llamada al método `manejarEventoActualizar`, "Eliminar" llama al método `manejarEventoEliminar`, "Registrar Tarjeta" llama al método `manejarEventoRegistrarTarjeta`, mientras que los botones "Servicio" y "Salir" son administrados por la superclase `Manejador` a través del método `manejarEventosAdicionales`. Estos métodos los describiremos con mayor detalles en breve.

```
  // Contrato 1: Manejar Evento
  public void manejarEvento(String str) {
      if (str.equals("Registrar"))
            manejarEventoRegistrar();
      else if (str.equals("Actualizar"))
            manejarEventoActualizar();
      else if (str.equals("Eliminar"))
            manejarEventoEliminar();
      else if (str.equals("Registrar Tarjeta"))
            manejarEventoRegistrarTarjeta();
      else
            manejarEventosAdicionales(str);
  }
```

El contrato "2" "Registrar Usuario" consta de tres responsabilidades, `crearRegistroUsuario`, `validarRegistroUsuario` y `obtenerRegistroUsuario`. El método `crearRegistroUsuario` se encarga de desplegar la pantalla `pantallaCrearRegUsuario`. Antes de solicitar su despliegue se verifica que la pantalla exista.

```
  // Contrato 2: Registrar Usuario
  public void crearRegistroUsuario() {
      if (pantallaCrearRegUsuario == null)
         pantallaCrearRegUsuario = new PantallaCrearRegUsuario(interfaceUsuario,this);
      desplegarPantalla(pantallaCrearRegUsuario);
  }
```

El método `validarRegistroUsuario` recibe dos parámetros del usuario, "login" y "contraseña", y se encarga de solicitar a la clase `InterfaceBaseDatosRegistro`, a través ed la referencia `interfaceRegistro`, que valide al usuario. El resultado debe ser un "verdadero" o "falso" ("true" o "false"). Nótese, que estamos enviando un objeto de tipo `RegistroUsuario` como parte de la llamada a la clase `InterfaceBaseDatosRegistro`. Esto realmente no es necesario, pero lo hacemos para aprovechar la llamada de validación a la base de datos, y obtener junto con la validación la información del usuario. De esta manera evitamos tener que hacer una segunda llamada para obtener los propios datos del usuario.

```
  public boolean validarRegistroUsuario(String log, String pass) {
       if (registroUsuario == null)
          registroUsuario = new RegistroUsuario();
       return interfaceRegistro.validarRegistro(registroUsuario,log,pass);
  }
```

El método `obtenerRegistroUsuario` se encarga de desplegar la pantalla `pantallaObtenerRegUsuario`. Antes de solicitar su despliegue se verifica que la pantalla exista, y que también exista un `RegistroUsuario` que contenga la información solicitada. Si ambos existen, se continúa escribiendo los datos del `RegistroUsuario` en la pantalla, mediante el método `escribirElementos` definido en la superclase `Manejador`. Continuaremos con el resto de los métodos de la clase `ManejadorRegistroUsuario` antes de explicar como escribir y leer datos de las pantallas.

```
  public void obtenerRegistroUsuario() {
      if (registroUsuario == null)
         System.out.println("Registro Invalido");
      else {
            if (pantallaObtenerRegUsuario == null)
                  pantallaObtenerRegUsuario
                        = new PantallaObtenerRegUsuario(interfaceUsuario,this);
            escribirElementos(pantallaObtenerRegUsuario,registroUsuario);
            desplegarPantalla(pantallaObtenerRegUsuario);
      }
  }
```

El método `escribirElementos` lo definimos en la clase `Manejador` (aunque mostrado recién aquí), el cual se encarga de solicitar a la pantalla que actualice sus campos de textos de acuerdo a los datos enviados. Más adelante explicaremos los detalles de esta escritura.

```
  protected void escribirElementos(Pantalla p,Datos datos) {
      p.escribirElementos(datos);
  }
```

El método `manejarEventoRegistrar` verifica que se haya instanciado un `RegistroUsuario` para luego guardar allí los datos insertados por el usuario en le pantalla, mediante el método `leerElementos`, definido en la

superclase `Manejador` (análogo a `escribirElementos`). Una vez obtenidos los datos y guardados en el `RegistroUsuario`, estos son enviados a la `InterfaceBaseDatosRegistro`, a través del método `crearRegistro`, para ser guardados en la base de datos. Finalemente, aprovechamos el método `obtenerRegistroUsuario` ya existente para desplegar la `PantallaObtenerRegUsuario` con los nuevos datos.

```
  private void manejarEventoRegistrar() {
       if (registroUsuario == null)
          registroUsuario = new RegistroUsuario();
       leerElementos(pantalla,registroUsuario);
       interfaceRegistro.crearRegistro(registroUsuario);
       obtenerRegistroUsuario();
  }
```

El método `leerElementos` lo definimos en la clase `Manejador` (aunque mostrado también aquí), el cual se encarga de solicitar a la pantalla que lea sus campos en el objeto de tipo datos enviados. Más adelante explicaremos los detalles de esta lectura.

```
  protected void leerElementos(Pantalla p,Datos datos) {
      p.leerElementos(datos);
  }
```

El método `manejarEventoActualizar` se comporta de manera similar a `registrarUsuario` en el sentido que se leen los elementos de la pantalla para luego actualizar la base de datos mediante el método `actualizarRegistro` (antes se creaba el registro). Se pudiera agregar la llamada `obtenerRegistroUsuario` al final, pero dado que ya se está en la pantalla `PantallaObtenerRegUsuario` no hay necesidad de hacerlo.

```
  private void manejarEventoActualizar() {
       if (registroUsuario == null)
          registroUsuario = new RegistroUsuario();
       leerElementos(pantalla,registroUsuario);
       interfaceRegistro.actualizarRegistro(registroUsuario);
  }
```

El método `manejarEventoEliminar` toma un `RegistroUsuario` ya existente, actualmente desplegado en la pantalla, y solicita a la `InterfaceBaseDatosRegistro` su eliminación mediante el método `eliminarRegistro`. Una vez eliminado el registro se regresa al flujo correspondiente a la creación de un nuevo registro, mediante la llamada `crearRegistroUsuario`.

```
  private void manejarEventoEliminar() {
       if (registroUsuario == null)
          System.out.println("Registro Invalido");
       else {
          interfaceRegistro.eliminarRegistro(registroUsuario);
          crearRegistroUsuario();
       }
  }
```

Finalmente, el método `manejarEventoRegistrarTarjeta` se encarga de solicitar al `ManejadorRegistroTarjeta` que procese el contrato "Registrar Tarjeta", correspondiente al método `registrarTarjeta` de éste último. Como parte de este método, primero se instancia el nuevo manejador para luego obtener un identificador correspondiente al usuario actual. Esto se hace mediante la llamada `leerValor(0)` del `RegistroUsuario`. El "0" corresponde al primer atributo del registro, en otras palabras el "login".

```
  private void manejarEventoRegistrarTarjeta() {
       if (registroUsuario == null)
          System.out.println("Registro Invalido");
       else {
          if (mrt == null)
              mrt = new ManejadorRegistroTarjeta(this,interfaceUsuario);
          String id = registroUsuario.leerValor(0);
          mrt.registrarTarjeta(id);
       }
  }
```

La descripción de las clases `PantallaRegUsuario`, `PantallaCrearRegUsuario` y `PantallaObtenerRegUsuario` es muy similar a la `PantallaPrincipal` anteriormente descrita.

Antes de proseguir, es importante añadir los métodos faltantes para la clase `Pantalla` (descrita antes de manera parcial). Los métodos áun pendientes de describir son `leerTexto`, `leerElementos` y `escribirElementos`.

El método `leerTexto` se muestra a continuación.

```
  public String leerTexto(String name0) {
    String name = null, str = null;
    for (int j = 0; name0.equals(name) == false && j < textos.size(); j++) {
          name = ((Component) textos.elementAt(j)).getName();
          if (name0.equals(name) == true)
            str = ((TextField) textos.elementAt(j)).getText();
    }
    return str;
  }
```

El método recibe un nombre, `name0`, correspondiente al campo que se desea leer de la pantalla. Recordemos que cada campo de texto en la pantalla fue asignado con un nombre. El ciclo del "for" compara este nombre contra la variable `name` mediante la llamada `name0.equals(name) == false`. Cuando estas dos variables son iguales el ciclo termina. También puede terminar, cuando se acaba de revisar todos los campos de texto en la pantalla, especificado mediante `textos.size()`. La variable `name` lee los diferentes nombres asignados a los campos de la pantalla, algo que se hace mediante la llamada:

```
          name = ((Component) textos.elementAt(j)).getName();
```

Se vuelve a hacer la comparación de nombres:

```
          if (name0.equals(name) == true)
```

Si los nombres son iguales, se prosigue leyendo el dato insertado por el usuario:

```
            str = ((TextField) textos.elementAt(j)).getText();
```

Dado que los campos coinciden, el ciclo del "for" se termina y se sale del método regresando el valor de `str` correspondiente al dato insetado pro el usuario en el campo correspondiente.

El método anterior fue diseñado para leer el dato correspondiente a un solo campo de la pantalla. Este método se extiende en el método `leerElementos` para leer todos los campos de manera automática, y guardarlos en un objeto de tipo `Datos`, en lugar de un `String`, en el caso anterior. Para ello, se pasa como parámetro, un objeto ya instanciado que defina atributos con nombres similares a aquellos que definen campos de texto en la pantalla. Esto es muy importante, dado que si no coinciden, no lograremos obtender ningún valor. Por otro lado, al ya estar instanciado el objeto, simplemente nos haría faltar rellenar los valores para los atributos correspondientes.

El método `leerElementos` se describe a continuación.

```
  public void leerElementos(Datos datos) {
    String name0,str,name;
    for (int i = 0; i < datos.numeroAtributos(); i++) {
      name0 = (String)datos.leerNombre(i);
      str = null;
      name = null;
      for (int j = 0; name0.equals(name) == false && j < textos.size(); j++) {
          name = ((Component) textos.elementAt(j)).getName();
          if (name0.equals(name) == true)  {
            str = ((TextField) textos.elementAt(j)).getText();
            datos.escribirValor(i,str);
          }
      }
    }
  }
```

El ciclo interno del "for" (el segundo ciclo) es exactamente igual al definido en el método `leerValor`. El cambio en este método radica en el "for" externo (el primer ciclo), donde se cicla por todos los atributos del objeto de tipo Datos hasta haber leído el nombre de todos los atributos mediante `datos.numeroAtributos()`. Durante este ciclo se lee el nombre de cada atributo:

```
      name0 = (String)datos.leerNombre(i);
```

Este nombre, `name0`, corresponde al nombre originalmente enviado como parámetro en `leerValor`. A diferencia del método anterior, aquí se cicla por todos los atributos, y en lugar de devolver el valor encontrado cunado exista una correspondencia entre campos, lo que hacemos es copiar el valor al objeto de tipo datos:

```
            datos.escribirValor(i,str);
```

El método `escribirValor` está definido en la clase `Datos` y ya fue descrito anteriormente.

El siguiente método `escribirElementos`, se encarga de hacer el opuesto al método anterior. El método `escribirElementos` recibe un objeto de tipo `Datos` y copia los valores de sus atributos a los campos correspondientes en la pantalla. Este método se muestra a continuación.

```
  public void escribirElementos(Datos datos) {
    String name0,str,name;
    for (int i = 0; i < datos.numeroAtributos(); i++) {
      name0 = (String)datos.leerNombre(i);
      str = (String)datos.leerValor(i);
      name = null;
      for (int j = 0; name0.equals(name) == false && j < textos.size(); j++) {
          name = ((Component) textos.elementAt(j)).getName();
          if (name0.equals(name) == true)
            ((TextField) textos.elementAt(j)).setText(str);
      }
    }
  }
```

Ambos ciclos son similares, existiendo únicamente dos modificaciones en la lógica. La primera, es que el valor de `str` se obtiene del atributo en lugar del campo de la pantalla:

```
      str = (String)datos.leerValor(i);
```

El segundo cambio es que se escribe del atributo a la pantalla en lugar de cómo se hacía antes:

```
            ((TextField) textos.elementAt(j)).setText(str);
```

Recuérdese que estos tres métodos anteriores son parte de la definición de la clase `Pantalla`. Ahora continuamos con las clase de registro de usuario.

La clase `RegistroUsuario` se mantiene igual a como se describió anteriormente en la Tabla 8.53, como se muestra en la Tabla 9.8.

Clase: RegistroUsuario	
Descripción: Para poder utilizar el sistema de reservaciones, el usuario debe estar registrado con el sistema. El registro contiene información acerca del usuario que incluye nombre, dirección, colonia, ciudad, país, código postal, teléfono de casa, teléfono de oficina, fax, email, login y password.	
Módulo: Registro.Usuario	
Estereotipo: Entidad	
Propiedades: Concreta	
Superclase: Datos	
Subclase:	
Atributos: login, password, nombre, apellido, dirección, colonia, ciudad, país, CP, telCasa, telOficina, fax, email	

Tabla 9.8. Tarjeta para la clase *RegistroUsuario* con responsabilidades, colaboraciones, jerarquías y contratos de actualizar y consultar información de registro para el caso de uso *RegistrarUsuario*.

La clase `RegistroUsuario` hereda de la superclase `Datos`.

```
public class RegistroUsuario extends Datos
```

Dentro del constructor de la clase inicializamos los atributos mediante llamadas a `agregarAtributo` definidos en la superclase `Datos`. El primer parámetro es el nombre del atributo, mientras que en el segundo inicializamos su valor, en este caso con cadenas vacías.

```
    public RegistroUsuario() {
       agregarAtributo("login","");
       agregarAtributo("password","");
       agregarAtributo("nombre","");
       agregarAtributo("apellido","");
       agregarAtributo("direccion","");
       agregarAtributo("colonia","");
       agregarAtributo("ciudad","");
       agregarAtributo("pais","");
       agregarAtributo("CP","");
       agregarAtributo("telCasa","");
       agregarAtributo("telOficina","");
       agregarAtributo("fax","");
       agregarAtributo("email","");
    }
```

No se define ningún método adicional para la clase `RegistroUsuario`.

Tarjeta

La clase *ManejadoRegistroTarjeta* se muestra en la Tabla 9.9.

Clase: ManejadorRegistroTarjeta	
Descripción: El manejador de registro de tarjeta se encarga de todo lo relacionado con registro de la tarjeta del usuario para poder pagar las reservaciones.	
Módulo: Registro.Tarjeta	
Estereotipo: Control	
Propiedades: Concreta	
Superclase: Manejador	
Subclase:	
Atributos: PantallaCrearRegTarjeta, PantallaObtenerRegTarjeta, RegistroTarjeta, InterfaceRegistro	
Contratos	
1. Manejar Evento	
manejarEvento(Evento) devuelve void *Método sobrescrito de la clase Manejador, encargado de recibir eventos del sistema de ventanas a través de la InterfaceUsuario.*	
2. Registrar Tarjeta	
registrarTarjeta(String) devuelve void *Método encargado de crear u obtener un RegistroTarjeta*	
Responsabilidades Privadas	
crearRegistroTarjeta() devuelve void *Método encargado de solicitar a la InterfaceBaseDatosRegistro la creación de un nuevo RegistroTarjeta a través del contrato de "Registrar Tarjeta"*	
obtenerRegistroTarjeta() devuelve void *Método encargado de solicitar a la InterfaceBaseDatosRegistro la obtención de un RegistroTarjeta a través del contrato de "Registrar Tarjeta"*	InterfaceBaseDatosRegistro (2)
manejarEventoRegistrar() devuelve void *Método encargado de solicitar a la InterfaceBaseDatosRegistro la creación de un nuevo RegistroTarjeta a través del contrato de "Registrar Tarjeta"*	InterfaceBaseDatosRegistro (2)
manejarEventoActualizar() devuelve void *Método encargado de solicitar a la InterfaceBaseDatosRegistro la actualización de un*	InterfaceBaseDatosRegistro (2)

RegistroTarjeta a través del contrato de "Registrar Tarjeta"	
manejarEventoEliminar() devuelve void *Método encargado de solicitar a la InterfaceBaseDatosRegistro la eliminación de un RegistroTarjeta a través del contrato de "Registrar Tarjeta"*	InterfaceBaseDatosRegistro (2)

Tabla 9.9. Tarjeta para la clase *ManejadoRegistroTarjeta* con responsabilidades, colaboraciones, jerarquías, contratos, subsistemas y protocolos identificadas del caso de uso *RegistrarTarjeta*.

La clase `ManejadorRegistroTarjeta` hereda de la superclase `Manejador`.

```
public class ManejadorRegistroTarjeta extends Manejador
```

Se definen los atributos especificados anteriormente, y también agregamos un nuevo atributo `idRegistro` de tipo `String` para guardar la referencia al "login" del usuario dueño del registro de tarjeta actualmente manipulado. Este nuevo atributo facilitará el manejo local de la información.

```
  private Pantalla pantallaCrearRegTarjeta;
  private Pantalla pantallaObtenerRegTarjeta;
  private RegistroTarjeta registroTarjeta;
  private InterfaceRegistro interfaceRegistro;
  private String idRegistro;
```

El constructor de la clase instancia un objeto de tipo `RegistroTarjeta` y recibe la referencia a la clase `InterfaceBaseDatosRegistro`. En caso de que la referencia sea nula, se instancia un nuevo objeto. Las pantallas son inicializadas durante la ejecución del programa.

```
  public ManejadorRegistroTarjeta(Manejador m,InterfaceUsuario ui) {
          super(m,ui);
      registroTarjeta = new RegistroTarjeta();
      interfaceRegistro = ((ManejadorRegistroUsuario) m).getInterfaceRegistro();
      if (interfaceRegistro == null)
               interfaceRegistro = new InterfaceBaseDatosRegistro();
  }
```

El contrato "1" "Manejar Evento" se encarga de administrar el comportamiento del registro de tarjeta dependiendo del botón que oprima el usuario. En el caso de "Registrar" se llama al método `manejarEventoRegistrar`, en el caso de "Actualizar" se llama al método `manejarEventoActualizar`, en el caso de "Eliminar" se llama al método `manejarEventoEliminar`, y en el caso de "Servicio" y "Salir" se llama al método `manejarEventosAdicionales`, el cual se encarga de darle manejo a estas dos opciones.

```
  // Contrato 1
  public void manejarEvento(String str) {
      if (str.equals("Registrar"))
          manejarEventoRegistrar();
      else if (str.equals("Actualizar"))
          manejarEventoActualizar();
      else if (str.equals("Eliminar"))
          manejarEventoEliminar();
      else
          manejarEventosAdicionales(str);
  }
```

El contrato "2" "RegistrarTarjeta" define un método `registrarTarjeta` el cual se describe a continuación.

```
  // Contrato 2
  public void registrarTarjeta(String log) {
      idRegistro = log;
      if (registroTarjeta == null)
         registroTarjeta = new RegistroTarjeta();
      boolean fg = interfaceRegistro.obtenerRegistro(registroTarjeta,log);
      if (fg == false)
         crearRegistroTarjeta();
      else
         obtenerRegistroTarjeta();
  }
```

El método recibe un parámetro, `log`, de tipo `String` correspondiente al usuario actualmente validado. Se verifica si ya existe un registro de tarjeta para el usuario mediante una llamada a la `InterfaceBaseDatosRegistro`:

```
      boolean fg = interfaceRegistro.obtenerRegistro(registroTarjeta,log);
```

Si aún no existe un registro de terjeta se solicita la creación de uno nuevo mediante el método `crearRegistroTarjeta`, de lo contrario se continúa con su despliegue mediante el método `obtenerRegistroTarjeta`.

El método `crearRegistroTarjeta` está encargado de desplegar la `PantallaCrearRegTarjeta`.

```
  private void crearRegistroTarjeta() {
      if (pantallaCrearRegTarjeta == null)
        pantallaCrearRegTarjeta = new PantallaCrearRegTarjeta(interfaceUsuario,this);
      desplegarPantalla(pantallaCrearRegTarjeta);
  }
```

El método `obtenerRegistroTarjeta` está encargado de escribir los datos obtenidos en el `registroTarjeta` en la `PantallaObtenerRegTarjeta` para luego ser desplegados.

```
  private void obtenerRegistroTarjeta() {
      if (registroTarjeta == null)
         System.out.println("Registro Invalido");
      else {
         if (pantallaObtenerRegTarjeta == null)
              pantallaObtenerRegTarjeta
                     = new PantallaObtenerRegTarjeta(interfaceUsuario,this);
         escribirElementos(pantallaObtenerRegTarjeta,registroTarjeta);
         desplegarPantalla(pantallaObtenerRegTarjeta);
      }
  }
```

El método `manejarEventoRegistrar` se encarga de leer los elementos de la pantalla y escribirlos en la base de datos. Al finalizar, se continúa con su despliegue en la `pantallaObtenerRegTarjeta` mediante la llamada `obtenerRegistroTarjeta`.

```
  private void manejarEventoRegistrar() {
      if (registroTarjeta == null)
         registroTarjeta = new RegistroTarjeta();
      registroTarjeta.escribirValor(0,idRegistro);
      leerElementos(pantalla,registroTarjeta);
      interfaceRegistro.crearRegistro(registroTarjeta);
      obtenerRegistroTarjeta();
  }
```

El método `manejarEventoActualizar` se encarga de leer los datos de la pantalla al `registroTarjeta` y actualizar la base de datos.

```
  private void manejarEventoActualizar() {
      if (registroTarjeta == null)
         registroTarjeta = new RegistroTarjeta();
      leerElementos(pantalla,registroTarjeta);
      interfaceRegistro.actualizarRegistro(registroTarjeta);
  }
```

El método `manejarEventoEliminar` se encarga de eliminar los datos actuales del `registroTarjeta` y solicitar su eliminación de la base de datos.

```
  private void manejarEventoEliminar() {
      if (registroTarjeta == null)
         System.out.println("Registro Invalido");
      else {
         interfaceRegistro.eliminarRegistro(registroTarjeta);
         crearRegistroTarjeta();
      }
  }
```

La descripción de las clases `PantallaRegUsuario`, `PantallaCrearRegUsuario` y `PantallaObtenerRegUsuario` es muy similar a la `PantallaPrincipal` anteriormente descrita.

La clase `RegistroTarjeta` se mantiene igual a como se describió anteriormente en la Tabla 8.53, como se muestra en la Tabla 9.7.

La tarjeta de clase para `RegistroTarjeta` se muestra en la Tabla 9.10.

Clase: RegistroTarjeta
Descripción: Para poder hacer un pago con una tarjeta de crédito, se debe tener un registro de tarjeta. El registro

contiene información acerca de la tarjeta incluyendo nombre, número, expedidor y vencimiento. La tarjeta está ligada a un registro de usuario.	
Módulo: Registro.Tarjeta	
Estereotipo: Entidad	
Propiedades: Concreta	
Superclase: Datos	
Subclase:	
Atributos: login, nombre, número, tipo, fecha	

Tabla 9.10. Tarjeta para la clase *RegistroTarjeta* con responsabilidades, colaboraciones, jerarquías y contratos de actualizar y consultar información de registro para el caso de uso *RegistrarTarjeta*.

La clase `RegistroTarjeta` hereda de la superclase `Datos`.

```
public class RegistroTarjeta extends Datos
```

Dentro del constructor de la clase inicializamos los atributos mediante llamadas a `agregarAtributo` definidos en la superclase `Datos`. El primer parámetro es el nombre del atributo, mientras que en el segundo inicializamos su valor, en este caso con cadenas vacías.

```
    public RegistroTarjeta() {
       agregarAtributo("login","");
       agregarAtributo("nombre","");
       agregarAtributo("numero","");
        agregarAtributo("tipo","");
       agregarAtributo("fecha","");
    }
```

No se define ningún método adicional para la clase `RegistroTarjeta`.

Interface Registro

La clase `InterfaceRegistro` se muestra en la Tabla 9.11.

Clase: InterfaceRegistro	
Descripción: Superclase para las interfaces a base de datos de registro y archivos.	
Módulo: Registro.InterfaceDB	
Estereotipo: Borde	
Propiedades: Abstracta	
Superclase:	
Subclase:	
Atributos:	
Contratos	
1. Registrar Usuario	
crearRegistro(RegistroUsuario) devuelve void *Método encargado de solicitar a la BaseDatosRegistro la creación de un nuevo RegistroUsuario*	BaseDatosRegistro
obtenerRegistro(RegistroUsuario) devuelve void *Método encargado de solicitar a la BaseDatosRegistro la obtención de un RegistroUsuario*	BaseDatosRegistro
actualizarRegistro(RegistroUsuario) devuelve void *Método encargado de solicitar a la BaseDatosRegistro la actualización de un RegistroUsuario*	BaseDatosRegistro
eliminarRegistro(RegistroUsuario) devuelve void *Método encargado de solicitar a la BaseDatosRegistro la eliminación de un RegistroUsuario*	BaseDatosRegistro
validarRegistro(String, String) devuelve bolean *Método encargado de solicitar a la BaseDatosRegistro la validación de un usuario*	BaseDatosRegistro
2. Registrar Tarjeta	

crearRegistro(RegistroTarjeta) devuelve void *Método encargado de solicitar a la BaseDatosRegistro la creación de un nuevo RegistroTarjeta*	BaseDatosRegistro
obtenerRegistro(RegistroTarjeta) devuelve void *Método encargado de solicitar a la BaseDatosRegistro la obtención de un RegistroTarjeta*	BaseDatosRegistro
actualizarRegistro(RegistroTarjeta) devuelve void *Método encargado de solicitar a la BaseDatosRegistro la actualización de un RegistroTarjeta*	BaseDatosRegistro
eliminarRegistro(RegistroTarjeta) devuelve void *Método encargado de solicitar a la BaseDatosRegistro la eliminación de un RegistroTarjeta*	BaseDatosRegistro

Tabla 9.11. Tarjeta para la clase *InterfaceRegistro* con responsabilidades, colaboraciones, jerarquías, contratos y protocolos de escribir y leer información de registro de usuario y registro de tarjeta para los casos de uso *RegistrarUsuario*, *ValidarUsuario* y *RegistrarTarjeta*.

Definimos la superclase `InterfaceRegistro` de la siguiente manera.

```
public abstract class InterfaceRegistro
```

Definimos un solo grupo de métodos para ambos contratos, generalizando el parámetro a `Datos` y especificando todos los métodos para que devuelvan un tipo booleano,

```
  // Contrato 1: Registrar Usuario
  // Contrato 2: Registrar Tarjeta
  public abstract boolean crearRegistro(Datos reg);
  public abstract boolean actualizarRegistro(Datos reg);
  public abstract boolean eliminarRegistro(Datos reg);
  public abstract boolean validarRegistro(Datos reg,String log,String pass);
  public abstract boolean obtenerRegistro(Datos reg,String log);
```

Agregamos un método que necesitaremos para lograr un manejo genérico de los diferentes objetos entidad. Este método se encargará de recibir como parámetro un objeto especializado devolviendo el nombre como `String` de su clase. Este nombre luego lo utilizaremos para instanciar nuevos objetos de esta clase de manera anónima, algo que en Java se facilita mucho.

```
  public String getClassName(Datos reg){
      String regname;
      Class regclass = reg.getClass();
      String fullname = regclass.getName();
      int index = fullname.lastIndexOf('.');
      if (index > 0)
          regname = fullname.substring(index+1);
      else
          regname = fullname;
      return regname;
  }
```

Se obtiene la clase de un objeto mediante la siguiente llamada:

```
      Class regclass = reg.getClass();
```

Luego obtenemos el nombre como cadena de esta clase,

```
      String fullname = regclass.getName();
```

Sin embargo, este nombre incluye el prefijo de todos los paquetes. Si deseamos únicamente obtener el nombre propio de la clase sin información sobre sus paquetes, lo que haríamos es obtener la posición del úlimo "." En el nombre,

```
      int index = fullname.lastIndexOf('.');
```

Finalmente buscamos el nombre a partir del último ".",

```
          regname = fullname.substring(index+1);
```

En el caso de que no incluya el nombre ningún paquete, lo devolveríamos completo,

```
          regname = fullname;
```

La clase *InterfaceBaseDatosRegistro* se muestra en la Tabla 9.12.

Clase: InterfaceBaseDatosRegistro
Descripción: La información de cada usuario se almacena en la base de datos de registro la cual se accesa mediante

la interface de la base de datos de registro. Esto permite validar a los distintos usuarios además de guardar información sobre la tarjeta de crédito para pagos en línea.	
Módulo: Registro.InterfaceDB	
Estereotipo: Borde	
Propiedades: Concreta	
Superclase:	
Subclase:	
Atributos:	
Contratos	
1. Registrar Usuario	
crearRegistro(RegistroUsuario) devuelve void *Método encargado de solicitar a la BaseDatosRegistro la creación de un nuevo RegistroUsuario*	BaseDatosRegistro
obtenerRegistro(RegistroUsuario) devuelve void *Método encargado de solicitar a la BaseDatosRegistro la obtención de un RegistroUsuario*	BaseDatosRegistro
actualizarRegistro(RegistroUsuario) devuelve void *Método encargado de solicitar a la BaseDatosRegistro la actualización de un RegistroUsuario*	BaseDatosRegistro
eliminarRegistro(RegistroUsuario) devuelve void *Método encargado de solicitar a la BaseDatosRegistro la eliminación de un RegistroUsuario*	BaseDatosRegistro
validarRegistro(String, String) devuelve bolean *Método encargado de solicitar a la BaseDatosRegistro la validación de un usuario*	BaseDatosRegistro
2. Registrar Tarjeta	
crearRegistro(RegistroTarjeta) devuelve void *Método encargado de solicitar a la BaseDatosRegistro la creación de un nuevo RegistroTarjeta*	BaseDatosRegistro
obtenerRegistro(RegistroTarjeta) devuelve void *Método encargado de solicitar a la BaseDatosRegistro la obtención de un RegistroTarjeta*	BaseDatosRegistro
actualizarRegistro(RegistroTarjeta) devuelve void *Método encargado de solicitar a la BaseDatosRegistro la actualización de un RegistroTarjeta*	BaseDatosRegistro
eliminarRegistro(RegistroTarjeta) devuelve void *Método encargado de solicitar a la BaseDatosRegistro la eliminación de un RegistroTarjeta*	BaseDatosRegistro

Tabla 9.12. Tarjeta para la clase *InterfaceBaseDatosRegistro* con responsabilidades, colaboraciones, jerarquías, contratos y protocolos de escribir y leer información de registro de usuario y registro de tarjeta para los casos de uso *RegistrarUsuario*, *ValidarUsuario* y *RegistrarTarjeta*.

La clase `InterfaceBaseDatosRegistro` se define a continuación:

```
public class InterfaceBaseDatosRegistro extends InterfaceRegistro
```

Se declaran tres atributos, como se explicó en el capítulo 5. La variable de tipo `Connection` se utiliza para hacer la conexión a la base dedatos, la variable de tipo `Statement` se utiliza para enviar la llamada de SQL a la base de datos, y la variable de tipo `ResultSet` para obtener los resultadas de la llamada a la base de datos.

```
    private Connection con;
    private Statement stmt;
    private ResultSet rs;
```

El constructor de la clase revisa mediante `revisarDriverSun` que exista la biblioteca con los "drivers" necesarios. Si estos existen se solicita abrir la conexión mediante la llamada `abrirConexion`. Los parámetros enviados a esta última llamada, en nuestro caso, son el nombre de la base de datos para poder identificarla en el

sistema (ver Apéndice con ejemplo de cómo configurar este nombre externamente), el “login” y “password” si es que estos han sido habilitados.
Para una aplicación de un sólo usuario se puede abrir la conexión a la base de datos al inicio de la aplicación, como lo estamos haciendo en nuestro ejemplo, para luego cerrarla al finalizar la aplicación. En el caso de aplicaciones con múltiples usuarios donde pueden existir accesos concurrentes a la base de datos, estas conexiones deben hacerse de manera dinámica, abriendo y cerrando las conexiones cada vez que sea haga una solicitud a la base de datos, de lo contrario se saturarían rápidamente las posibles conexiones a la base de datos.

```
    public InterfaceBaseDatosRegistro()
    {
        if (checkDriverSun() == 0)
            abrirConexion("jdbc:odbc:reservaciones", "alfredo", "ITAM");
    }
```

El método `revisarDriverSun` revisa que existan las bibliotecas adecuadas.

```
    private int revisarDriverSun()
    {
        try {
            Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
        }
        catch (ClassNotFoundException ex) {
            return -1;
        }
        return 0;
    }
```

El método `abrirConexion` hace la conexión a la base de datos, algo que fue explicado anteriormente en el Capítulo 5.

```
    private void abrirConexion(String url,String log,String pass)
    {
        try {
          con = DriverManager.getConnection (url, log, pass);
          stmt = con.createStatement();
        }
        catch (SQLException ex) {
             ...
        }
    }
```

Como parte de los contratos “1” y “2”, “Registrar Usuario” y “Registrar Tarjeta”, respectivamente, definimos un solo conjunto de métodos tomando como parámetro la superclase `Datos`, en lugar de los tipos especializados.
El método crearRegistro se encarga de insertar nuevos registros a la base de datos, el cual se muestra a continuación.

```
    public boolean crearRegistro(Datos reg)
    {
        String regname = getClassName(reg);
        String textsql = reg.serializarSQLinsert();
        String querysql = "INSERT INTO " + regname + " " + textsql;
        return actualizarRecordSetRegistro(querysql);
    }
```

El método recibe un registro, `reg`, del cual obtiene el nombre de su clase, cuyo nombre deberá corresponder al nombre de la tabla correspondiente en la base de datos.

```
        String regname = getClassName(reg);
```

Dado que la llamada a la base de datos se hace mediante una llamada en SQL, es necesario generar esta llamada como cadena. Aprovechamos que todas las clases entidad heredan de `Datos` para especificar un método `serializarSQLinsert` a nivel de la superclase, que se encargue de generar el formato de texto deseado por SQL. La llamada es la siguiente,

```
        String textsql = reg.serializarSQLinsert();
```

Por ejemplo, el texto devuelto por `serializarSQLinsert` a partir de un nuevo registro de usuario sería similar al siguiente:

```
(login, password, rpassword, nombre, apellido, direccion, colonia, ciudad, pais, CP, telCasa, telOficina, fax, email) VALUES ('alfredo', 'awr', 'awr', 'Alfredo', 'Weitzenfeld', 'Río Hondo #1', 'San Angel Tizapán', 'México DF', 'México', '01000', '56284000', '56284000 x3614', '56162211', 'alfredo@itam.mx')
```

Una vez obtenido el texto con los datos de la llamada, se prosigue a componer la llamada completa,

```
        String querysql = "INSERT INTO " + regname + " " + textsql;
```

En este caso el nombre de la tabla será `RegistroUsuario`, similar al de la clase, y se formaría la llamada de SQL completa,

```
INSERT INTO RegistroUsuario (login, password, rpassword, nombre, apellido, direccion, colonia, ciudad, pais, CP, telCasa, telOficina, fax, email) VALUES ('alfredo', 'awr', 'awr', 'Alfredo', 'Weitzenfeld', 'Río Hondo #1', 'San Angel Tizapán', 'México DF', 'México', '01000', '56284000', '56284000 x3614', '56162211', 'alfredo@itam.mx')
```

El método `serializarSQLinsert` se define en la clase `Datos` y no es complicado, simplemente toma los nombres y valores de los atributos para generar la lista de textos, como se muestra a continuación,

```
  public String serializarSQLinsert() {
         String serializa0 = "";
         String serializa1 = "";
         for (int i = 0; i < campos[1].size(); i++) {
                if (i > 0)   {
                  serializa0 = serializa0 + ", ";
                  serializa1 = serializa1 + ", ";
                }
                serializa0 = serializa0 + campos[0].elementAt(i);
                serializa1 = serializa1 + "'" + campos[1].elementAt(i) + "'";
         }
         return "(" + serializa0 + ") VALUES (" + serializa1 + ")";
  }
```

El método genera una cadena de texto `serializa0` con los nombres de los atributos divididos por comas, y otra cadena de texto, `serializa1`, con los valores para lso atributos, también separados por comas. En el caso de valores de texto, como en nuestro ejemplo, estos valores son limitados por comillas sencillas.

Finalmente, la llamada `actualizarRecordSetRegistro` hace la propia llamada a la base de datos.

```
    private boolean actualizarRecordSetRegistro(String query)
    {
       try {
            int n = stmt.executeUpdate (query);
            return true;
       }
       catch (SQLException ex) {
             ...
       }
       return false;
  }
```

Este método consiste principalmente de la llamada `stmt.executeUpdate` la cual recibe el "query" en forma de `String` y devuelve un entero correspondiente al número de récords que fueron insertados o actualizados correctamente.

El método para actualizar información `actualizarRegistro` de cierta tabla es muy similar al anterior, con la diferencia que se basa en un dato existente.

```
    public boolean actualizarRegistro(Datos reg)
    {
        String log = reg.leerValor(0);
        String regname = getClassName(reg);
        String textsql = reg.serializarSQL();
        String str = "UPDATE " + regname + " SET " + textsql +
             " WHERE Login = '" + log + "';";
        return actualizarRecordSetRegistro(str);
    }
```

El método recibe un registro, `reg`, del cual obtiene el nombre de su clase, cuyo nombre deberá corresponder nuevamente al nombre de la tabla correspondiente en la base de datos. Se generará nuevamente una llamada en SQL, sólo que algo diferente de la anterior. Nuevamente aprovechamos que todas las clases entidad heredan de `Datos`

para especificar un método `serializarSQL` a nivel de la superclase, que se encargue de generar el formato de texto deseado por SQL. La llamada es la siguiente,

```
        String textsql = reg.serializarSQL ();
```

Por ejemplo, el texto devuelto por `serializarSQL` a partir de un nuevo registro de usuario sería similar al siguiente:

```
login = 'alfredo', password = 'awr', nombre = 'Alfredo', apellido = 'Weitzenfeld', direccion = 'Río Hondo #1', colonia = 'San Angel Tizapán', ciudad = 'México DF', pais = 'México', CP = '01000', telCasa = '56284000', telOficina = '56284000 x3614', fax = '56162211', email = 'alfredo@itam.mx'
```

Una vez obtenido el texto con los datos de la llamada, se prosigue a componer la llamada completa,

```
        String str = "UPDATE " + regname + " SET " + textsql +
             " WHERE Login = '" + log + "';";
```

En este caso el nombre de la tabla será `RegistroUsuario`, similar al de la clase, y se formaría la llamada de SQL completa,

```
UPDATE RegistroUsuario SET login = 'alfredo', password = 'awr', nombre = 'Alfredo', apellido = 'Weitzenfeld', direccion = 'Río Hondo #1', colonia = 'San Angel Tizapán', ciudad = 'México DF', pais = 'México', CP = '01000', telCasa = '56284000', telOficina = '56284000 x3614', fax = '56162211', email = 'alfredo@itam.mx' WHERE login = 'alfredo'
```

El método `serializarSQL` se implementa en la clase `Datos` de manera muy similar al método `serializarSQLinsert`.

```
  public String serializarSQL() {
    String serializa = "";
    for (int i = 0; i < campos[1].size(); i++) {
      if (i > 0)
        serializa = serializa + ", ";
      serializa = serializa + campos[0].elementAt(i) + " = " +
        "'" + campos[1].elementAt(i) + "'";
    }
    return serializa;
  }
```

A diferencia de `serializarSQLinsertar`, mostramos un formato distinto donde los nombres de los atributos van inmediatamente seguidos por sus valores, por lo cual una sola cadena de texto será suficiente.

La llamada final es al método `actualizarRecordSetRegistro` de manera similar al método anterior de `crearRegistro`.

El método `eliminarRegistro` es más sencillo que los anteriores, ya que no requiere de ninguna serialización de datos.

```
    public boolean eliminarRegistro(Datos reg)
    {
        String log = reg.leerValor(0);
        String regname = getClassName(reg);
        String str = "DELETE FROM " + regname + " WHERE Login = '" + log + "';";
        return actualizarRecordSetRegistro(str);
    }
```

Se obtiene el "login" del usuario mediante `reg.leerValor(0)`, el cual es luego utilizado para generar la eliminación del registro.

```
        String str = "DELETE FROM " + regname + " WHERE Login = '" + log + "';";
```

Un ejemplo de la cadena para SQL sería,

```
DELETE FROM RegistroUsuario WHERE Login = 'alfredo'
```

La llamada a `actualizarRecordSetRegistro` es similar a los métodos anteriores.

El método `validarRegistro` se encarga de validar al usuario y al mismo tiempo obtener un registro.

```
    public boolean validarRegistro(Datos reg,String log, String pass) {
        String regname = getClassName(reg);
        String querysql = "SELECT * FROM " + regname +
              " WHERE (login = '" + log + "') AND (password = '" + pass + "')";
        return leerRecordSetRegistro(querysql,reg);
    }
```

Se obtiene el nombre de la clase correspondiente a la tabla deseada para luego generar la llamada con un "query", como por ejemplo,

```
SELECT * FROM RegistroUsuario WHERE (login = 'alfredo' AND password = 'awr')
```

Una vez generada la llamada de SQL, el resultado se se llama al método `leerRecordSetRegistro`, el cual se muestra a continuación,

```
    private boolean leerRecordSetRegistro(String query,Datos datos)
    {
        ResultSetMetaData rsmd;
        String str;

        try {
            rs = stmt.executeQuery(query);
            rsmd = rs.getMetaData();
            if  (rs != null && rs.next()) {
                for (int i = 1; i <= rsmd.getColumnCount(); i++) {
                    str = rs.getString(i);
                    datos.escribirValor(i-1,str);
                }
                return true;
            }
        }
        catch (SQLException ex) {
             ...
        }
        return false;
    }
```

A diferencia de las demás llamadas anteriroes, que utilizan un `executeUpdate`, la consulta utiliza un `executeQuery`, la cual devuelve una lista de objetos, referida por `rs` y de tipo `ResultSet`. Si el valor al cual se refiere `rs` fuese nulo, esto significaría que no se encontró ningún objeto correspondiente a la búsqueda recién hecha. La información "meta" del objeto `rs` puede obtenerse mediante la llamada `rs.getMetaData()` para obtener, por ejemplo, el número de columnas definidas en la tabla correspondiente a `rs` y a su vez correspondiente a los atributos del objeto de tipo `Datos`.

Los propios datos son obtenidos a través de la llamada `rs.next()`, la cual obtiene el siguiente dato en la lista de datos. Si se continúa haciendo llamadas similares se obtendría datos adicionales en la lista, si estos existen. Por lo tanto, la llamada puede ponerse dentro de un "while" en lugar de un "if", en el caso de múltiples datos. Dentro del ciclo "for" se obtiene cada atributo delo objeto mediante,

```
                    str = rs.getString(i);
```

Este valor, en nuestro caso una cadena, es luego escrita al propio objeto de tipo Datos mediante,

```
                    datos.escribirValor(i-1,str);
```

En el caso de múltiples objetos que se obtuvieran de la consulta, será necesario tener acceso o instanciar múltiples objetos de tipo `Datos`, en lugar del actualmente único objeto llamado `datos`.

El método para obtener información `obtenerRegistro` de cierta tabla es similar al anterior aunque más sencillo ya que no incluye la validación de la contraseña pero sí la información sobre el usuario, como se muestra a continuación.

```
    public boolean obtenerRegistro(Datos reg,String log)
    {
        String regname = getClassName(reg);
        String querysql = "SELECT * FROM " + regname +
              " WHERE (login = '" + log + "')";
        return leerRecordSetRegistro(querysql,reg);
    }
```

A continuación describimos la clase `InterfaceArchivoRegistro`, como se muestra en la Tabla 9.13.

Clase: InterfaceArchivoRegistro

Descripción: La información de cada usuario se almacena en la base de datos de registro la cual se accesa mediante la interface de la base de datos de registro. Esto permite validar a los distintos usuarios además de guardar información sobre la tarjeta de crédito para pagos en línea.	
Módulo: Registro.InterfaceDB	
Estereotipo: Borde	
Propiedades: Concreta	
Superclase:	
Subclase:	
Atributos:	
Contratos	
1. Registrar Usuario	
crearRegistro(RegistroUsuario) devuelve void *Método encargado de solicitar a la BaseDatosRegistro la creación de un nuevo RegistroUsuario*	ArchivoRegistro
obtenerRegistro(RegistroUsuario) devuelve void *Método encargado de solicitar a la BaseDatosRegistro la obtención de un RegistroUsuario*	ArchivoRegistro
actualizarRegistro(RegistroUsuario) devuelve void *Método encargado de solicitar a la BaseDatosRegistro la actualización de un RegistroUsuario*	ArchivoRegistro
eliminarRegistro(RegistroUsuario) devuelve void *Método encargado de solicitar a la BaseDatosRegistro la eliminación de un RegistroUsuario*	ArchivoRegistro
validarRegistro(String, String) devuelve bolean *Método encargado de solicitar a la BaseDatosRegistro la validación de un usuario*	ArchivoRegistro
2. Registrar Tarjeta	
crearRegistro(RegistroTarjeta) devuelve void *Método encargado de solicitar a la BaseDatosRegistro la creación de un nuevo RegistroTarjeta*	ArchivoRegistro
obtenerRegistro(RegistroTarjeta) devuelve void *Método encargado de solicitar a la BaseDatosRegistro la obtención de un RegistroTarjeta*	ArchivoRegistro
actualizarRegistro(RegistroTarjeta) devuelve void *Método encargado de solicitar a la BaseDatosRegistro la actualización de un RegistroTarjeta*	ArchivoRegistro
eliminarRegistro(RegistroTarjeta) devuelve void *Método encargado de solicitar a la BaseDatosRegistro la eliminación de un RegistroTarjeta*	ArchivoRegistro

Tabla 9.13. Tarjeta para la clase *InterfaceArchivoRegistro* con responsabilidades, colaboraciones, jerarquías, contratos y protocolos de escribir y leer información de registro de usuario y registro de tarjeta para los casos de uso *RegistrarUsuario*, *ValidarUsuario* y *RegistrarTarjeta*.

La clase `InterfaceArchivoRegistro` hereda de la superclase `InterfaceRegistro`.

```
public class InterfaceArchivoRegistro extends InterfaceRegistro
```

Definimos un número de atributos privados, como son la dirección de la ubicación de los archivos, junto con varaibles temporales que serán descritas más adelante.

```
  private String path = "reservaciones/baseDatos";
  private Datos reg;
  private String  regname;
  private File freg, ftar;
  private Vector archivoRegistro;
  private ArchivoRegistro ar;
```

El constructor se sencarga de inicializar la leída de los archivos. Aquí haremos elgo que no es muy eficiente, pero en el caso de archivos pequeños no significa gran costo. Nos referimso al hecho de leer todos los archivos completos a

memoria para administrar la información más fácilmente. Recuérdese que esto sería prohibitivo en el caso de grandes archivos, lo cual habría que leer en secciones limitadas según las necesidades particulares del momento.
En este caso, dado que se trata de dos archivos relacionados con registro, `RegistroUsuario.dat` y `RegistroTarjeta.dat`, correspondiente a las dos tablas anterioremente descritas, lo que haremos será leerlos inicialmente a memoria, como se describe continuación.

```
public InterfaceArchivoRegistro()
{
     archivoRegistro = new Vector();
     reg = new RegistroUsuario();
     regname = getClassName(reg);
     ar = new ArchivoRegistro(path,reg,regname);
     archivoRegistro.addElement(ar);
     reg = new RegistroTarjeta();
     regname = getClassName(reg);
     ar = new ArchivoRegistro(path,reg,regname);
     archivoRegistro.addElement(ar);
}
```

Al prinicpio, instanciamos un objeto `archivoRegistro` de tipo `Vector`, donde guardaremos objetos de tipo `ArchivoRegistro`, cada uno de ellos correspondiente a un tipo de datos de registro. A continuación instanciamos un objeto de tipo `RegistroUsuario` y obtenemos el nombre de la clase, como le hicimos anteriormente. Hecho esto, instanciamos el propio objeto `ArchivoRegistro`, donde en el caso de un RegistroUsuario, los datos se verían como a continuación,

```
      ar = new ArchivoRegistro("reservaciones/baseDatos",reg,"RegistroUsuario");
```

Este método lee del archivo y carga sus contenidos a memoria como veremos más adelante cuando describamos esta clase. Por cada archivo que se lee, agregamos una referencia al vector `archivoRegistro`.
Hacemos lo mismo para el archivo que guarda información de `RegistroTarjeta`.
El método `obtenerRegistro` es el encargado de obtener datos sobre un usuario. Esto se hace obteniendo primero el nombre de la clase, como se hizo enteriormente.

```
public boolean obtenerRegistro(Datos reg, String log)
{
     String regname = getClassName(reg);
     for (int i = 0; i < archivoRegistro.size(); i++) {
           ar = (ArchivoRegistro) archivoRegistro.elementAt(i);
           if (ar != null && regname.equals(ar.getName()) == true)
                 return ar.leerRegistro(reg, log);
     }
     return false;
}
```

Se hace una búsqueda de los diferentes archivos leidos en memoria dependiendo del tipo de información que se busca. El ciclo del "for" pasa por todos estos objetos, dos en nuestro caso (`RegistroUsuario` y `RegistroTarjeta`) y compara su nombre con el tipo deseado,

```
           if (ar != null && regname.equals(ar.getName()) == true)
```

Una vez que concuerdan el nombre de la clase con el nombre del tipo de archivo en memoria, se copia los datos del archivo en memoria a los del registro meidante la llamada al método `leerRegistro` el cual será descrito más adelante, el cual es llamado de la siguiente forma,

```
                 return ar.leerRegistro(reg, log);
```

El método `crearRegistro`, es similar al anterior, encargándose de primero buscar el archivo correcto para luego hacer la llamada al `archivoRegistro` correcto.

```
public void crearRegistro(Datos reg)
{
     String regname = getClassName(reg);
     for (int i = 0; i < archivoRegistro.size(); i++) {
           ar = (ArchivoRegistro) archivoRegistro.elementAt(i);
           if (ar != null && regname.equals(ar.getName()) == true)
                 ar.crearRegistro(reg);
     }
}
```

El método `crearRegistro` hace una llamada interna al método `crearRegistro` de la clase `ArchivoRegistro`.

De manera similar , el método `actualizarRegistro` hace una llamada al método `actualizarRegistro` del `ArchivoRegistro` correspondiente.

```
public void actualizarRegistro(Datos reg)
{
     String regname = getClassName(reg);
     for (int i = 0; i < archivoRegistro.size(); i++) {
           ar = (ArchivoRegistro) archivoRegistro.elementAt(i);
           if (ar != null && regname.equals(ar.getName()) == true)
                 ar.actualizarRegistro(reg);
            }
}
```

De manera similar `eliminarRegistro` se encarga de hacer la llamada adecuada al `ArchivoRegistro`.

```
public void eliminarRegistro(Datos reg)
{
     String regname = getClassName(reg);
     for (int i = 0; i < archivoRegistro.size(); i++) {
           ar = (ArchivoRegistro) archivoRegistro.elementAt(i);
           if (ar != null && regname.equals(ar.getName()) == true)
                 ar.eliminarRegistro(reg);
     }
}
```

Finalmente, el método `validarRegistro` compara los valores del usuario a través del método del mismo nombre en la clase `ArchivoRegistro`.

```
public boolean validarRegistro(Datos reg, String log, String pass)
{
     String regname = getClassName(reg);
     for (int i = 0; i < archivoRegistro.size(); i++) {
           ar = (ArchivoRegistro) archivoRegistro.elementAt(i);
           if (ar != null && regname.equals(ar.getName()) == true)
                 return ar.validarRegistro(reg, log, pass);
     }
     return false;
}
```

La clase ArchivoRegistro de se muestra en la Tabla 9.14.

Clase: ArchivoRegistro	
Descripción: La información de cada usuario se almacena en la base de datos de registro la cual se accesa mediante la interface de la base de datos de registro. Esto permite validar a los distintos usuarios además de guardar información sobre la tarjeta de crédito para pagos en línea.	
Módulo: Registro.InterfaceDB	
Estereotipo: Borde	
Propiedades: Concreta	
Superclases:	
Subclases:	
Atributos:	
Contratos	
1. Registrar Usuario	
crearRegistro(RegistroUsuario) devuelve void *Método encargado de solicitar a la BaseDatosRegistro la creación de un nuevo RegistroUsuario*	BaseDatosRegistro
obtenerRegistro(RegistroUsuario) devuelve void *Método encargado de solicitar a la BaseDatosRegistro la obtención de un RegistroUsuario*	BaseDatosRegistro
actualizarRegistro(RegistroUsuario) devuelve void *Método encargado de solicitar a la BaseDatosRegistro la actualización de un RegistroUsuario*	BaseDatosRegistro
eliminarRegistro(RegistroUsuario) devuelve void *Método encargado de solicitar a la BaseDatosRegistro la eliminación de un RegistroUsuario*	BaseDatosRegistro

validarRegistro(String, String) devuelve bolean *Método encargado de solicitar a la BaseDatosRegistro la validación de un usuario*	BaseDatosRegistro
2. Registrar Tarjeta	
crearRegistro(RegistroTarjeta) devuelve void *Método encargado de solicitar a la BaseDatosRegistro la creación de un nuevo RegistroTarjeta*	BaseDatosRegistro
obtenerRegistro(RegistroTarjeta) devuelve void *Método encargado de solicitar a la BaseDatosRegistro la obtención de un RegistroTarjeta*	BaseDatosRegistro
actualizarRegistro(RegistroTarjeta) devuelve void *Método encargado de solicitar a la BaseDatosRegistro la actualización de un RegistroTarjeta*	BaseDatosRegistro
eliminarRegistro(RegistroTarjeta) devuelve void *Método encargado de solicitar a la BaseDatosRegistro la eliminación de un RegistroTarjeta*	BaseDatosRegistro

Tabla 9.14. Tarjeta para la clase *InterfaceArchivoRegistro* con responsabilidades, colaboraciones, jerarquías, contratos, protocolos, atributos y algoritmos para los casos de uso *RegistrarUsuario*, *ValidarUsuario* y *RegistrarTarjeta*.

La clase `ArchivoRegistro` se muestra a continuación,

```
public class ArchivoRegistro
```

Definimos un número de atributos privados, entre los cuales la `listaRegistro` guarda la lista de todos los récords del archivo leido, uan referencia al archivo, incluyendo su nombre, junto con otras variables temporales.

```
  private Vector listaRegistro;
  private File freg;
  private Datos registro;
  private Class creg;
  private String name;
  private String filename;
```

Dado que cada `ArchivoRegistro` se encarga de un solo archivo, aprovechamos esta clase para hacer la inicialziación correspondiente. Leemos el nombre de la clase `classname` para obtener el nombre del archivo, al cual agregaremos la terminación “dat”. Se inicializa la `listaRegistro`, la referencia al archivo mediante `freg`, el cual se obtiene a través del `dirname` y `filename`. Finalmente hacemos la propia lectura de los archivos mediante la llamada `inicializarRegistrosArchivo`.

```
  public ArchivoRegistro(String dirname, Datos reg, String classname)
  {
       creg = reg.getClass();
       name = classname;
       filename = classname + ".dat";
       listaRegistro = new Vector();
      freg = new File(dirname,filename);
      inicializarRegistrosArchivo();
  }
```

En el método `inicializarRegistrosArchivo` hacemos la lectura de los registros mediante la obtención de un `BufferedReader` a partir de la referencia del archivo `freq`. La variable `is` junto con la `listaRegistro`, hasta el momento vacía, son enviados como parámetros al método `leerRegistrosArchivo`. Una vez leídos los registro se cierra el archivo.

```
  private void inicializarRegistrosArchivo()
  {
      try
      {
            BufferedReader is = new BufferedReader(new FileReader(freg));
            leerRegistrosArchivo(listaRegistro,is);
            is.close();
      }
      catch(IOException e)
      {
            ...
      }
  }
```

El método leerRegistrosArchivo se encarga de hacer la propia lectura, algo que se explicó anteriormente en el Capítulo 5.

```
  private void leerRegistrosArchivo(Vector vectorDatos, BufferedReader is)
      throws IOException
  {
      String s = is.readLine();
      Integer ns = Integer.valueOf(s);
      int n = ns.intValue();
      for (int i = 0; i < n; i++)
      {
            try {
                  registro = (Datos) creg.newInstance();
            }
            catch (Exception ex){
                  ...
            }
            s = is.readLine();
            StringTokenizer t = new StringTokenizer(s, "|");
            for (int j = 0; j < registro.numeroAtributos(); j++)
            {
                  String val = t.nextToken();
                  registro.escribirValor(j,val);
            }
            vectorDatos.addElement(registro);
      }
  }
```

Antes de explicar la lectura, mostramos un ejemplo del archivo correspondiente al `RegistroUsuario`,

```
2
alfredo|awr         |Alfredo|Weitzenfeld|Rio         Hondo         #1|San         Angel
Tizapan|Mexico|MEXICO|01000|6284000|6284000 x3614|6162211|alfredo@itam.mx|
reservas|awr        |Sistema|Reservaciones|Rio        Hondo        #1|San        Angel
Tizapan|Mexico|MEXICO|01000|6284000|6284000 x3614|6162211|alfredo@itam.mx|
```

La primera línea del archivo especifica el número de registros guardados en el archivo.

De manera similar, se muestra un ejemplo de archivo correspondiente al `RegistroTarjeta`.

```
2
alfredo|Alfredo Weitzenfeld|123456789|MasterCard|01/05|
reservas|Alfredo Weitzenfeld|987654321|Visa|02/4|
```

En este caso, cada registro de usuario cuenta con un registro de tarjeta correspondiente.

Ambos tipos de archivos son procesados de la siguiente forma,

```
      String s = is.readLine();
      Integer ns = Integer.valueOf(s);
      int n = ns.intValue();
```

Las últimas dos líneas convierten la cadena “2” referida por `s` a un número entrero.

Posteriormente se hace un ciclo “for” que ejecutará por el número de registros que hayan, y en cada ciclo instanciará un nuevo registro donde guardará los datos,

```
                  registro = (Datos) creg.newInstance();
```

Hecho esto, se continuará con la lectura de cada registro mediante,

```
            s = is.readLine();
```

Como se puede notar, en el archivo se separan los diversos campos mediante "|" (líneas verticales). Para ello definimos un "string tokenizer" en Java de la siguiente forma,

```
            StringTokenizer t = new StringTokenizer(s, "|");
```

Dentro del ciclo "for" se van leyendo las diversas especificadas entre las líneas verticales mediante,

```
                    String val = t.nextToken();
```

El valor obtenido, `val`, es luego copiado al registro mediante,

```
                    registro.escribirValor(j,val);
```

Dado que pueden haber múltiples registros (en nuestro caso ejemplo habían dos), se prosigue guardando el registro obtenido en el vector de datos,

```
            vectorDatos.addElement(registro);
```

A continuación escribimos los métodos que dan servicio a los contratos con nombre similar en la clase `InterfaceArchivoRegistro`. Comenzamos con `leerRegistro` encargado de leer a un registro temporal `reg` los datos guardados en memoria correspondiente al usuario especificado por `log`, como se muestra a continuación,

```
  public boolean leerRegistro(Datos reg, String log)
  {
      for (int i = 0; i < listaRegistro.size(); i++) {
             Datos datos = (Datos) listaRegistro.elementAt(i);
             if (log.equals(datos.leerValor(0))) {
                 for (int j = 0; j < datos.numeroAtributos(); j++)
                    reg.escribirValor(j,datos.leerValor(j));
                  return true;
             }
      }
      return false;
  }
```

El método revisa todos los elementos de la `listaRegistro` hasta obtener un registro cuyo nombre corresponda al especificado en `log`. Una vez encontrado el registro, los datos de este son copiados al registro, especificado por `reg`, mediante la llamada `escribirValor`.

El método `actualizarRegistro` se encarga de recibir un dato recién modificado y cambiarlo primero en la lista de registros en memoria para luego hacer la modificación correspondiente del archivo de registro.

```
  public void actualizarRegistro(Datos reg)
  {
      int indice = leerIndiceRegistro(reg.leerValor(0));
      if (indice  != -1) {
             listaRegistro.setElementAt(reg,indice);
              actualizarArchivoRegistro();
      }
  }
```

El método primero revisa que exista el registro correspondiente en memoria mediante la llamada,

```
      int indice = leerIndiceRegistro(reg.leerValor(0));
```

Una vez revisado que exista el registro mediante un `índice` correspondiente, la `listaRegistro` se actualzia con el nuevo registro,

```
            listaRegistro.setElementAt(reg,indice);
```

Finalmente, se actualiza el archivo correspondiente mediante,

```
             actualizarArchivoRegistro();
```

El método `leerIndiceRegistro` obtiene un registro por nombre, regname, y devuelve el registro cuyo nombre corresponda al solicitado. Una vez encontrado el registro se devuelve su índice.

```
  private int leerIndiceRegistro(String regname)
  {
       for (int i = 0; i < listaRegistro.size(); i++) {
              registro = (Datos) listaRegistro.elementAt(i);
              if (regname.equals(registro.leerValor(0)) == true)
                     return i;
       }
       return -1;
  }
```

El método `actualizarArchivoRegistro` es el opuesto a `leerArchivoRegistro`. En lugar de leer un archivo, el archivo se sobrescribe. Esto se hace cada vez que exista una modificación en algún registro en memoria. Este método llama al método `escribirDatos`.

```
  private void actualizarArchivoRegistro()
  {
       try
       {
              BufferedWriter os = new BufferedWriter(new FileWriter(freg));
              escribirDatos(listaRegistro, os);
              os.close();
       }
       catch(IOException e)
       {
                     ...
       }
  }
```

El método `escribirDatos` es el contrario de `leerDatos` anteriormente descrito.

```
   private void escribirDatos(Vector vectorDatos, BufferedWriter os)
       throws IOException
  {
       int num = vectorDatos.size();
       String numStr = String.valueOf(num);
       os.write(numStr);
       os.newLine();
       for (int i = 0; i < num; i++) {
              Datos datos = (Datos) vectorDatos.elementAt(i);
               for (int j = 0; j < datos.numeroAtributos(); j++) {
                     String str = datos.leerValor(j);
                     os.write(str+"|");
              }
              os.newLine();
       }
  }
```

Se lee el tamaño del vector de registros,

```
       int num = vectorDatos.size();
```

Este número se convierte a una cadena,

```
       String numStr = String.valueOf(num);
```

La cadena se escribe como una línea completa en el archivo,

```
       os.write(numStr);
       os.newLine();
```

A continuación se obitenen los datos del registro,

```
              Datos datos = (Datos) vectorDatos.elementAt(i);
```

Estos datos se copian uno por uno al archivo separándolos con una línea vertical,

```
                     String str = datos.leerValor(j);
                     os.write(str+"|");
```

Finalmente se pasa a la siguiente línea para continuar con otro registro como parte del ciclo “for”,

```
              os.newLine();
```

El método `eliminarRegistro` es muy similar a `actualizarRegistro`, con la diferencia de eliminar primero el registro de la lista para luego actualziar el archivo correspondiente, como se muestra a continuación.

```
public void eliminarRegistro(Datos reg)
{
    int indice = leerIndiceRegistro(reg.leerValor(0));
    if (indice  != -1) {
          listaRegistro.removeElementAt(indice);
           actualizarArchivoRegistro();
    }
}
```

El método `validarRegistro` se encarga de leer el registro de la memoria según el `log` especificado, mediante la llamada `leerRegistro`, para luego comparar si el `pass` corresponde, como se muestra a continuación.

```
public boolean validarRegistro(Datos reg, String log, String pass)
{
    if (leerRegistro(reg,log) != false && reg != null) {
        String preg = reg.leerValor(1);
        if (pass.equals(preg))
           return true;
    }
    return false;
}
```

Servicios

La tarjeta de clase para la clase `ManejadorServicio` se muestra en la Tabla 9.15.

Clase: ManejadorServicio	
Descripción: La información de cada usuario se almacena en la base de datos de registro la cual se accesa mediante la interface de la base de datos de registro. Esto permite validar a los distintos usuarios además de guardar información sobre la tarjeta de crédito para pagos en línea.	
Módulo: Servicios	
Estereotipo: Control	
Propiedades: Concreta	
Superclase: Manejador	
Subclase:	
Contratos	
1. Manejar Evento	
manejarEvento(Evento) devuelve void *Método sobrescrito de la clase Manejador, encargado de recibir eventos del sistema de ventanas a través de la InterfaceUsuario.*	
2. Ofrecer Servicio	
ofrecerServicio() devuelve void *Método encargado de hacer solicitudes de consultar, reservar y administración de registros*	
Responsablidades Privadas	
registrar() devuelve void *Método encargado de hacer la solicitud de administración de registros al SubsistemaRegistro*	SubsistemaRegistro (2)

Tabla 9.15. Tarjeta para la clase *ManejadorServicio* con responsabilidades, colaboraciones, jerarquías, contratos, subsistemas y protocolos a partir de los casos de uso *RegistrarUsuario* y *RegistrarTarjeta*.

La clase `ManejadorServicio` se define de la siguiente manera,

```
public class ManejadorServicio extends Manejador
```

De manera similar a los demás manejadores, la clase `ManejadorServicio` debe manejar los siguientes eventos de usuario correspondientes al contrato “1” de “Manejar Evento”,

```
// Contrato 1: Manejar Evento
public void manejarEvento(String str) {
  if (str.equals("Consultar Informacion"))
    consultar();
  else if (str.equals("Hacer Reservacion"))
    reservar();
  else if (str.equals("Obtener Registro"))
    registrar();
  else
    manejarEventosAdicionales(str);
}
```

No describiremos aquí el manejo de consultar ni reservar, aunque si el de registrar, como se puede ver a continuación,

```
private void registrar() {
    if (mru == null)
      mru = new ManejadorRegistroUsuario(this,interfaceUsuario);
    mru.obtenerRegistroUsuario();
}
```

El método `registrar()` se encarga de llamar al método `obtenerRegistroUsuario()` perteneciente al contrato "2" de la clase `RegistroUsuario`.
El contrato "2" "Ofrecer Servicio" se encarga de desplegar la `PantallaServicio`, como se muestra a continuación,

```
// Contrato 2: Ofrecer Servicio
public void ofrecerServicio() {
    desplegarPantalla(new PantallaServicio(interfaceUsuario,this));
}
```

La descripción de la clase `PantallaServicio` se mantiene de manera similar a las demás pantallas.

9.2 Diagrama de Clases

Una vez finalizada la especificación de la programación se procede a generar los diagramas de clase para el sistema completo. Este diagrama servirá como parte del apoyo visual al proceso de programación.
A continuación se muestran estos diagramas para el sistema de reservaciones de vuelo.

InterfaceUsuario

El diagrama de clases para las clases del módulo *InterfaceUsuario* se muestran en la Figura 9.1.

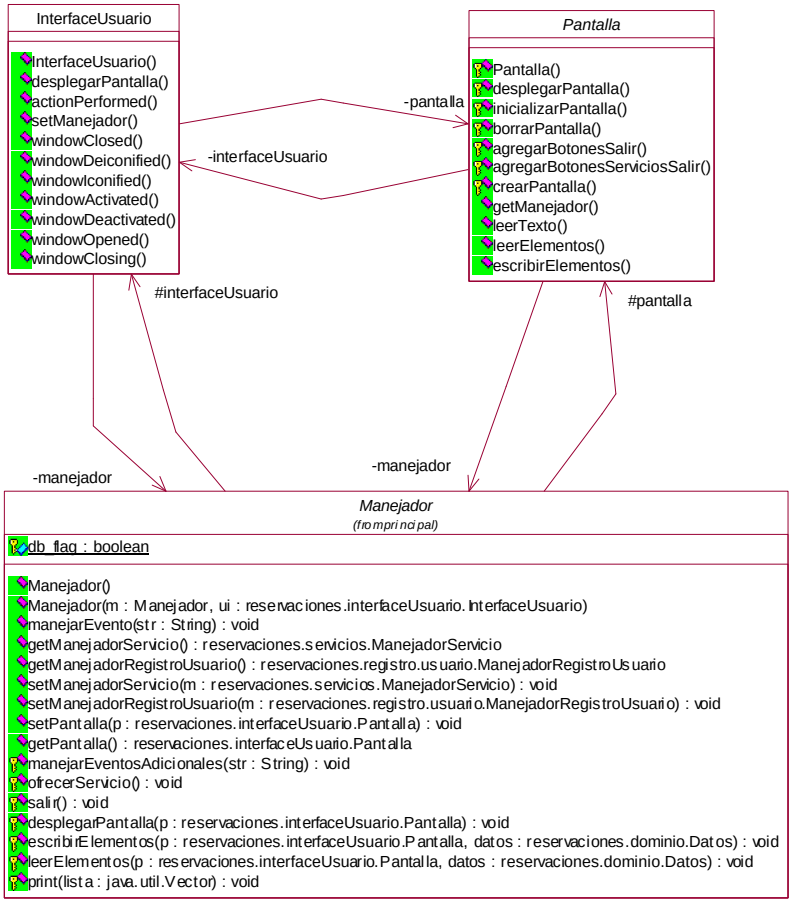

Figura 9.1. Diagrama de clases para las clases del módulo *InterfaceUsuario*.

Principal

El diagrama de clases para las clases del módulo *Principal* se muestran en la Figura 9.2.

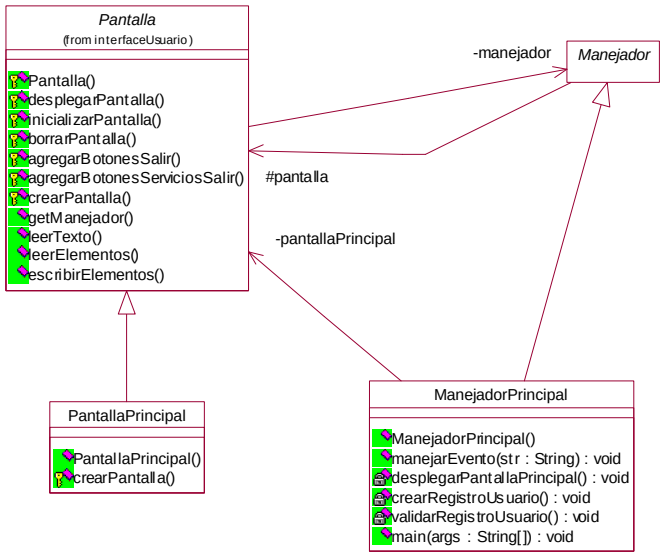

Figura 9.2. Diagrama de clases para las clases del módulo *Principal*.

Registro

El módulo de *Registro* se compone de los módulos de *Usuario*, *Tarjeta* y *InterfaceBD*, como se muestran en las siguientes secciones.

Usuario

El diagrama de clases para las clases del módulo *Usuario* se muestran en la Figura 9.3.

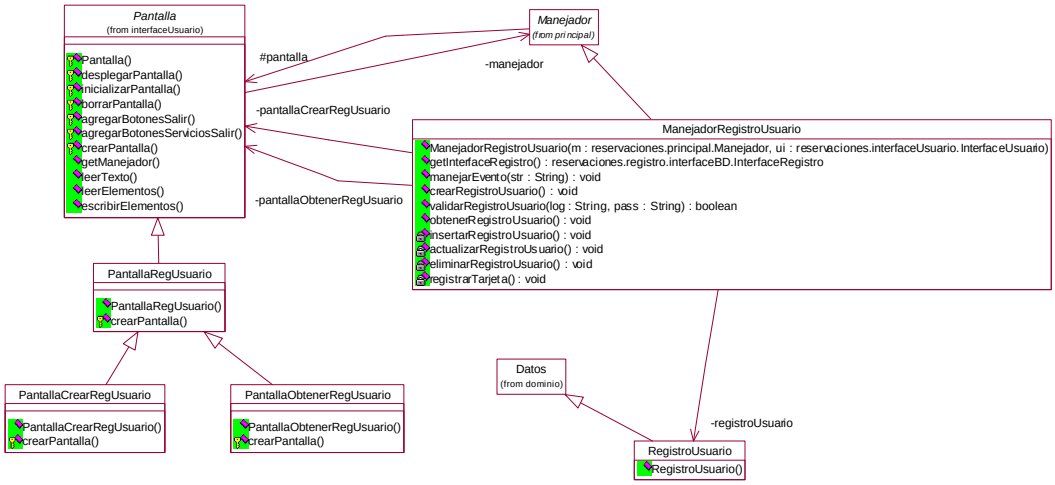

Figura 9.3. Diagrama de clases para las clases del módulo *Usuario*.

Tarjeta

El diagrama de clases para las clases del módulo *Tarjeta* se muestran en la Figura 9.4.

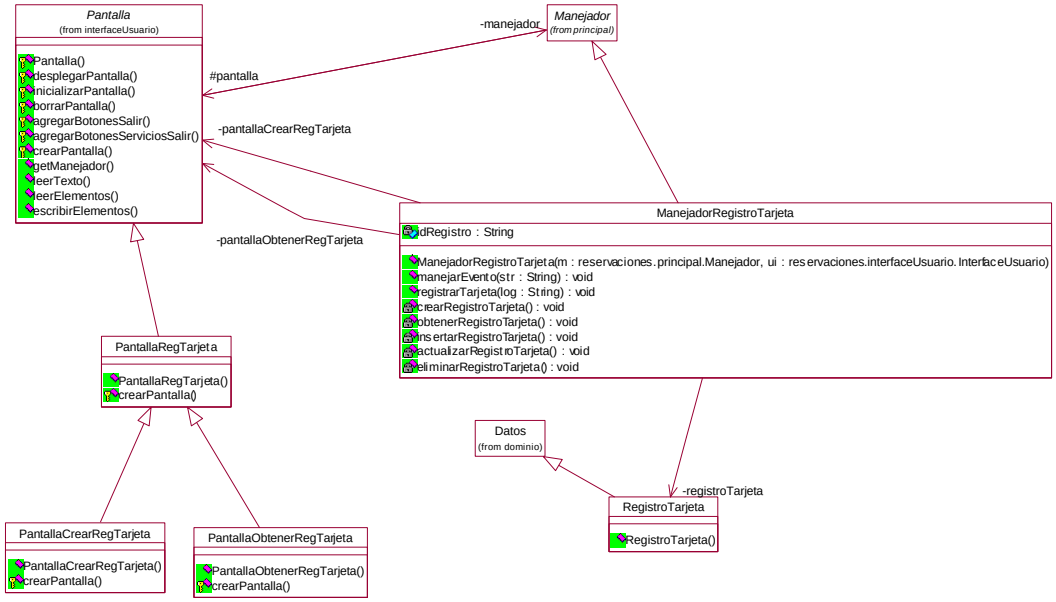

Figura 9.4. Diagrama de clases para las clases del módulo *Tarjeta*.

InterfaceBD

El diagrama de clases para las clases del módulo *InterfaceBD* se muestran en la Figura 9.5.

Figura 9.5. Diagrama de clases para las clases del módulo *InterfaceBD*.