ÍNDICE

Planteamiento del problema: actores participantes, relaciones entre actores, funcionalidad a cumplir por la práctica a desarrollar.

- Actores participantes.

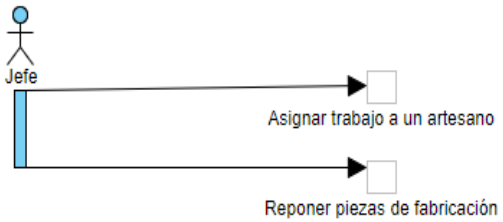
Un actor representa un rol externo al sistema y que interactúa con él, en nuestra práctica cada rol de usuario que puede realizar unas acciones propias son los siguientes:

Jefe, cliente, comercial, artesano, y un usuario de gestión, al que denominamos admin.

Los cuales son descritos a continuación:

JEFE

Un actor con el rol de jefe podrá realizar las siguientes acciones:



- Asignar trabajo a un artesano.

- Reponer piezas de fabricación.

Asignar trabajo a un artesano

Un cliente realiza un pedido, un comercial le otorga un presupuesto y si el cliente acepta el presupuesto se debe de comenzar con la producción de los muebles que forman el pedido.
Aquí es donde entra un usuario con el rol de jefe, que podrá asignar un pedido cuyo presupuesto ha sido aceptado y no esta asignado a ningún otro artesano, por lo tanto no se ha comenzado con su fabricación.
El artesano tendrá mas trabajo en su listado de tareas y deberá comenzar con la fabricación de estas.

Reponer piezas de fabricación

Un mueble está formado por distintas piezas, y si no existieran piezas para finalizar la fabricación de un mueble el usuario deberá detener la producción de este mueble, alegando que piezas le faltan.
Un usuario con rol de jefe será el encargado de reponer las piezas de fabricación de los muebles, para que los artesanos puedan retomar y finalizar los pedidos sin que esto afecte a la cadena de producción.

CLIENTE

Un actor con rol de cliente podrá realizar las siguientes operaciones:

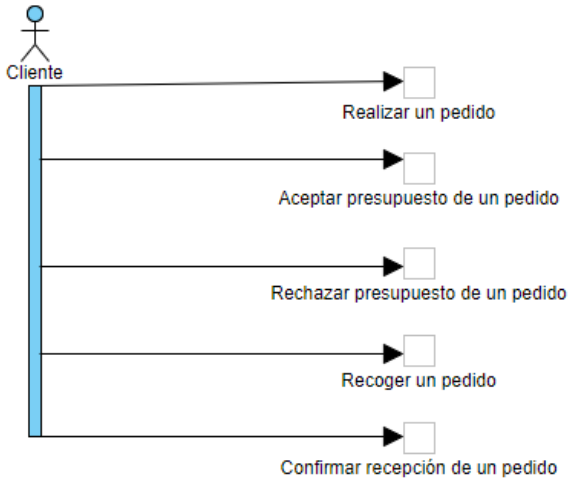
- Realizar un pedido.

- Aceptar presupuesto de un pedido.

- Rechazar presupuesto de un pedido.

- Recoger un pedido.

- Confirmar recepción de un pedido.



Realizar un pedido

Un usuario con el rol de cliente puede realizar un pedido, para ello deberá indicar que muebles deben formarlo, un pedido debe contener al menos un mueble.
En el momento que un cliente realiza un pedido este queda pendiente de que un comercial le asigne un presupuesto.

Aceptar presupuesto de un pedido

Una vez un cliente haya realizado un pedido y un comercial le haya asignado un presupuesto el cliente podrá aceptar su presupuesto, confirmando así la orden de su pedido quedando este pendiente de fabricación.
Un cliente solo podrá aceptar el presupuesto de un pedido que haya sido realizado por el y cuyo presupuesto esté pendiente de aceptación o de rechazar.

Rechazar presupuesto de un pedido

Una vez un cliente haya realizado un pedido y un comercial le haya asignado un presupuesto el cliente podrá rechazar su presupuesto, cancelando así la producción de este.
Un cliente solo podrá rechazar el presupuesto de un pedido que haya sido realizado por el y cuyo presupuesto esté pendiente de aceptación o no.

Recoger un pedido

Una vez un cliente haya realizado un pedido, haya aceptado el presupuesto de este y haya sido fabricado podrá recoger el pedido en la fábrica y finalizar por completo el pedido.
Un cliente solo podrá recoger un pedido que haya sido realizado por el y que esté pendiente de su recogida.

Confirmar recepción de un pedido

Una vez un cliente haya realizado un pedido, haya aceptado el presupuesto de este y haya sido fabricado y enviado podrá confirmar la recepción del pedido y finalizarlo por completo.
Un cliente solo podrá confirmar la recepción de un pedido un pedido que haya sido realizado por el y que esté pendiente de su confirmación.

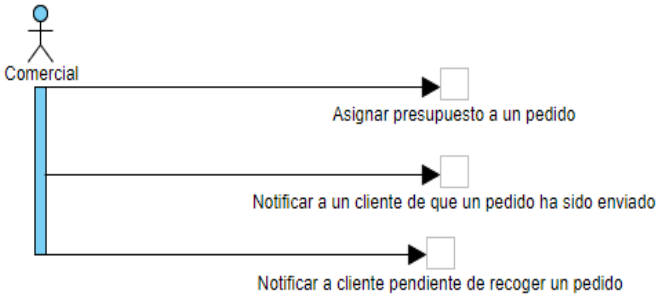
COMERCIAL

Un actor con rol de comercial podrá realizar las siguientes operaciones:

- Asignar presupuesto a un pedido.

- Notificar a un cliente que su pedido ha sido enviado.

- Notificar a cliente pendiente de recoger un pedido.



Asignar presupuesto a un pedido

Un usuario con rol de comercial podrá asignar un presupuesto a un pedido que figure como nuevo en el sistema, una vez asignado queda reflejado en el sistema y pasa a estar pendiente de su estudio por parte del cliente que lo ha realizado.

Notificar a un cliente que su pedido ha sido enviado

Un usuario con rol de comercial podrá cambiar el estado de un pedido a ‘enviado’ cuya fabricación haya sido completada por un artesano.
Una vez asignado queda reflejado en el sistema y pasa a estar pendiente de confirmación de recepción por parte del cliente que lo ha realizado.

Notificar a cliente pendiente de recoger un pedido

Un usuario con rol de comercial podrá cambiar el estado de un pedido a ‘pendiente de recogida’ cuya fabricación haya sido completada por un artesano.
Una vez asignado queda reflejado en el sistema y pasa a estar pendiente de su recogida por parte del cliente que lo ha realizado.

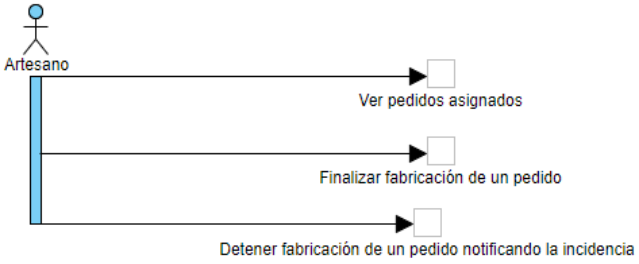
ARTESANO

Un actor con rol de artesano podrá realizar las siguientes operaciones:

- Ver pedidos asignados.

- Finalizar fabricación de un pedido.

- Detener fabricación de un pedido notificando la incidencia.



Ver pedidos asignados

Un usuario con rol de artesano podrá ver que pedidos tiene asignado, para ver un registro de las tareas que tiene pendientes de fabricación, las que están paradas y pendientes de piezas o las tareas que ha finalizado.
Un artesano solo podrá acceder a los pedidos que el tiene asignado.

Finalizar fabricación de un pedido

Un usuario con rol de artesano podrá finalizar la fabricación de un pedido, quedando este pendiente de envío o de recogida por parte del cliente, que antes debe ser informado por el comercial encargado.
Una vez finalizada la fabricación de un pedido el artesano se pondrá con la producción de la siguiente tarea pendiente, en el caso de que tenga algún trabajo pendiente.

Detener fabricación de un pedido notificando la incidencia

Un usuario con rol de artesano podrá detener la fabricación de un pedido alegando el motivo y que piezas necesita, el pedido pasará a estar en estado ‘parado’ y pendiente de un usuario con rol jefe reponga las piezas.

ADMIN

Un actor con rol de administrador podrá realizar las siguientes operaciones:

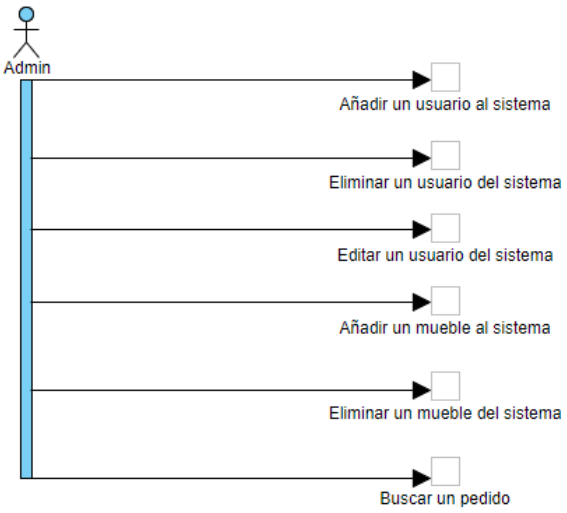
- Añadir un usuario al sistema.

- Eliminar un usuario del sistema.

- Editar un usuario del sistema.

- Añadir un mueble al sistema.

- Eliminar un mueble del sistema.



- Buscar un pedido.

Añadir un usuario al sistema

Un usuario de gestión se encargará de la gestión de usuarios del sistema, esto implica entre otros dar de alta a un usuario que podrá ser un cliente particular o empresa, o bien un empleado con rol de jefe, comercial o artesano (por hora o en plantilla).
Cada rol podrá tener atributos únicos a la hora de darlo de alta en el sistema.

Eliminar un usuario del sistema

Un usuario de gestión se encargará de la gestión de usuarios del sistema, esto implica entre otros eliminar a un usuario que podrá ser un cliente particular o empresa, o bien un empleado con rol de jefe, comercial o artesano (por hora o en plantilla).
El usuario de gestión deberá introducir un identificador único del usuario a eliminar y si este existe será eliminado.

Editar un usuario del sistema

Un usuario de gestión se encargará de la gestión de usuarios del sistema, esto implica entre otros editar un usuario que podrá ser un cliente particular o empresa, o bien un empleado con rol de jefe, comercial o artesano (por hora o en plantilla).
El usuario de gestión deberá introducir un identificador único del usuario a editar y sus nuevos datos, en el caso de que el usuario exista.

Añadir un mueble al sistema

Un usuario de gestión se encargará de la gestión de muebles del sistema, esto implica entre otros añadir un mueble que podrá ser un tipo de silla, como silla de cocina, silla de oficina (silla de oficina con ruedas y silla de oficina sin ruedas) y silla plegable, o bien un mueble de tipo mesa como mesa de café (mesa de café de cristal y mesa de café de madera), mesa de dormitorio y mesa de comedor.

Eliminar un mueble del sistema

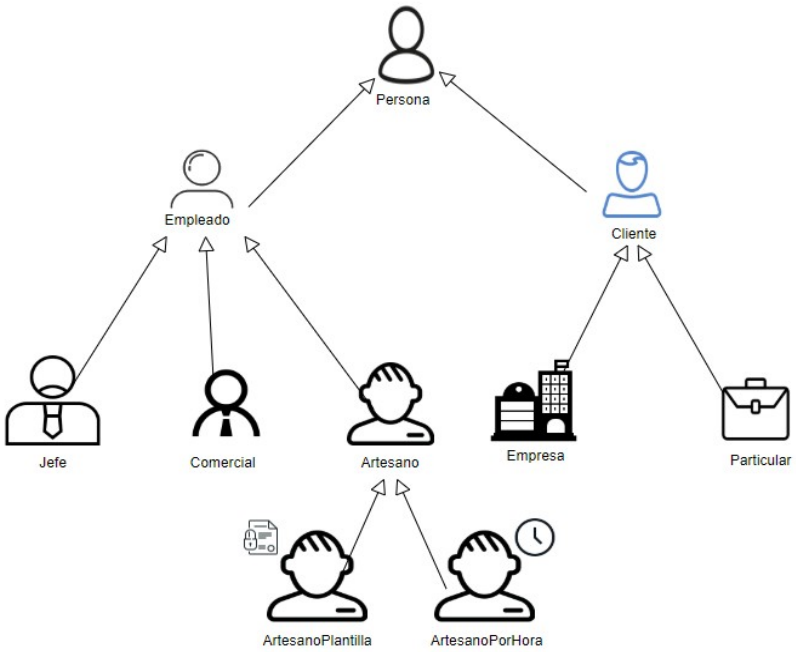
Un usuario de gestión se encargará de la gestión de muebles del sistema, esto implica entre otros eliminar un mueble que este en ‘stock’ en el sistema.
El mueble podrá ser un tipo de silla, como silla de cocina, silla de oficina (silla de oficina con ruedas y silla de oficina sin ruedas) y silla plegable, o bien un mueble de tipo mesa como mesa de café (mesa de café de cristal y mesa de café de madera), mesa de dormitorio y mesa de comedor.

Buscar un pedido

Un usuario de gestión tendrá acceso a todos los pedidos que figuran en el sistema y podrá realizar búsquedas sobre estos para verlos en detalle.

- Relaciones entre actores.

Relación de generalización.



- Funcionalidad a cumplir por la práctica a desarrollar.

La práctica a desarrollar consiste en diseñar e implementar un sistema integrado de gestión de una fábrica artesanal de muebles, con el fin de monitorizar y automatizar todo el proceso de fabricación, así como para la venta y la facturación de muebles artesanales producidos bajo demanda.

En general, a grandes rasgos, son varias las funciones que tiene un sistema de gestión de una fábrica artesanal de muebles:

• Recepción de un pedido de un cliente con los muebles que quiere comprar y las características de cada uno.

• Asignación por parte del jefe de un pedido a un artesano en la fábrica. Para simplificar la gestión del proceso en esta práctica, se da por hecho que todos los artesanos en la fábrica son igual de capaces de fabricar todos los muebles.

• Proceso de fabricación: un artesano, al terminar la fabricación de un mueble, acude al sistema para ver los próximos trabajos que le tocan.
Según la información en el sistema por el pedido actual, tiene que ponerse con la fabricación de un nuevo mueble y dejar constancia del proceso en el sistema, de manera que el jefe pueda inspeccionar los progresos de cada artesano y asignar los nuevos pedidos en consecuencia.

Si por el motivo que sea (por ejemplo, falta de piezas), un artesano deja un trabajo en un estado sin completar, anota el motivo en el sistema y pasa al siguiente trabajo.

• Gestión de usuarios: altas, bajas, modificaciones de las personas que figuran en el sistema, tanto empleados como clientes.

• Gestión de clientes por parte del comercial: comunicar a los clientes el precio de un pedido, informarles de que su pedido está listo para recoger/entregar, etc.

A estas funciones debemos añadir las siguientes:

• Realizar búsquedas sencillas sobre los empleados, clientes y pedidos.

• Permitir la asignación de las fichas de los empleados, clientes y pedidos.

• Permitir que cada artesano vea las fichas que le toca gestionar y pueda editar los datos dejando constancia del trabajo realizado y el estado de fabricación. Por ejemplo, pendiente, en proceso, parado (hace falta piezas, pendiente de confirmación del cliente), pendiente de recepción, terminado.

• Producir diferentes listados del funcionamiento de la fabrica: las piezas que hace falta pedir para las fabricaciones, las fichas procesadas por cada artesano, las confirmaciones que hay que solicitar a los clientes, las fichas en proceso y un historial de cada artesano y cada mueble.

ANEXO I.

ANÁLISIS Y JUSTIFICACIÓN DE DISEÑO

ESTRUCTURA DE PAQUETES

El código fuente de nuestro programa lo forman 3 clases (Menu, Prueba y Fabrica) y dos paquetes principales (modelo y servicio). Hemos ido estructurando en paquetes las distintas clases de este proyecto proporcionando las siguientes ventajas:

· Agrupamiento de clases con características comunes.

· Reutilización de código al promover principios de programación orientada a objetos como la encapsulación y modularidad.

· Mayor seguridad al existir niveles de acceso.

· Evita la colisión de clases que tengan el mismo nombre. Pueden existir clases con el mismo nombre siempre y cuando su fully qualified class name sean únicos.

· Mantenibilidad de código. Si un paquete se enfoca en la agrupación de clases con características comunes, el cambio en la funcionalidad se limita a las clases contenidas en dicho paquete, además, si es un paquete grande soporta la reusabilidad, si por el contrario es pequeño soporta su mantenibilidad.

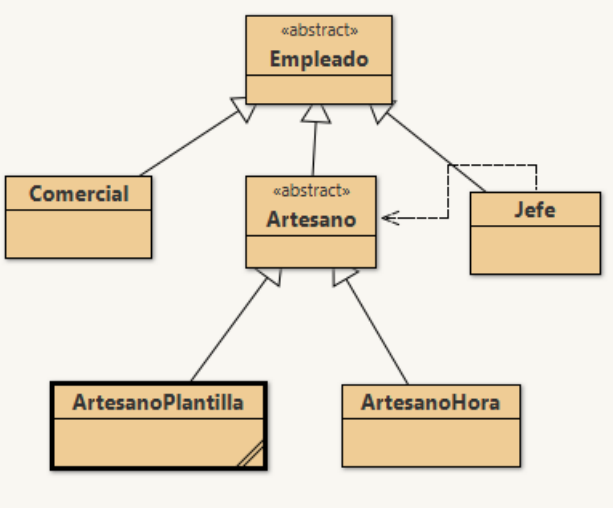
A continuación detallamos la estructura de nuestro proyecto:

Modelo

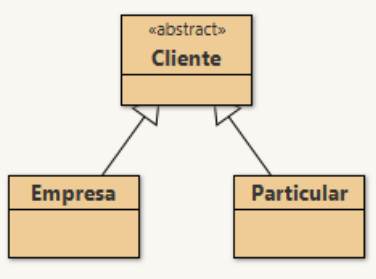
El **package modelo** contiene todas a las clases que representan entidades en el sistema.
Está compuesto por cuatro subpaquetes (persona, mueble, pedido y enumerable).

- **package persona**: aquí encontramos todo el modelado de persona o subclases que extienden de persona. Encontramos una clase abstracta (Persona) y dos subpaquetes (empleado y cliente).

  - **package empleado**: Encontramos todas las subclases que heredan de persona y representan entidades de empleados. En ella encontramos seis clases y forman la siguiente jerarquía:

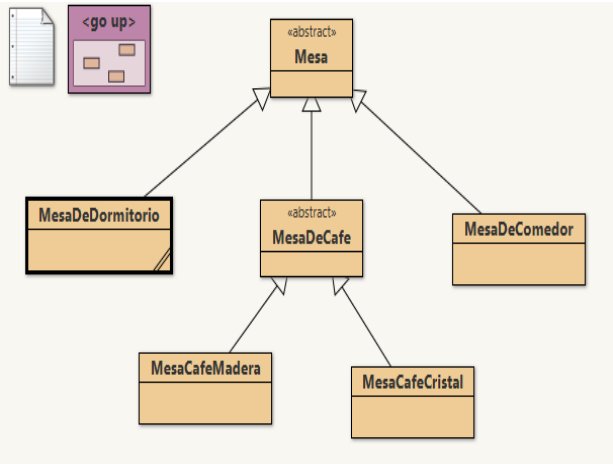


- **package cliente**: Encontramos todas las subclases que heredan de persona y representan entidades de clientes. En ella encontramos tres clases y forman la siguiente jerarquía:

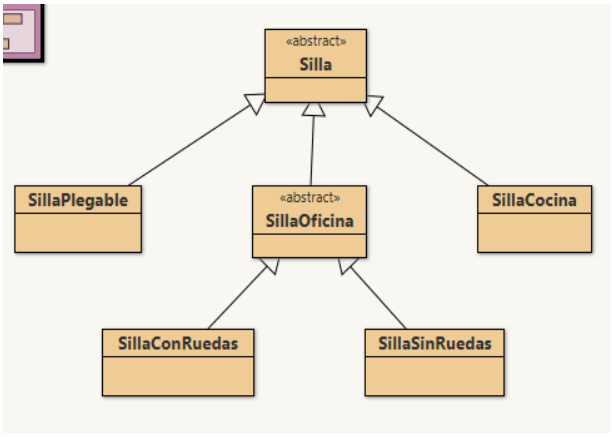


- **package mueble**: aquí encontramos todo el modelado de muebles o subclases que extienden de muebles. Encontramos una clase abstracta (Mueble) y dos subpaquetes (mesa y silla).

- **package mesa**: Encontramos todas las subclases que heredan de mesa y representan entidades de mesa. En ella encontramos seis clases y forman la siguiente jerarquía:



- **package silla**: Encontramos todas las subclases que heredan de silla y representan entidades de silla. En ella encontramos seis clases y forman la siguiente jerarquía:



- **package pedido**: En este subpaquete encontramos una única clase Pedido que contiene el modelado de la entidad pedido.
- **package enumerable**: Aqui encontramos los tipos enumerables que usamos en nuestra aplicación. Este paquete esta formados por las clases Estado, Operacion y Pieza:

· *Estado*: Listado de todos los estados disponibles en los que se puede encontrar un pedido.
· *Operacion*: Opreaciones CRUD disponibles para la gestión de datos. Crear, editar o eliminar.
· *Pieza*: Piezas que forman la fabricación de un mueble.

Servicio

En el package **servicio** encontramos las clases encargadas de la entrada, la salida y la gestión de los datos del sistema:

· *Salida*: Clase encargada de mostrar información al usuario por consola.
· *Entrada*: Clase encargada de la lectura de datos del sistema.
· *RegistroPedido*: Clase encargada de almacenar, insertar los pedidos que figuran en el sistema.
· *RegistroEmpleado*: Clase encargada de almacenar, insertar, editar y eliminar los empleados que figuran en el sistema.
· *ResgistroCliente*: Clase encargada de almacenar, insertar, editar y eliminar los clientes que figuran en el sistema.
· *RegistroMueble*: Clase encargada de almacenar, insertar, editar y eliminar los muebles que figuran en el sistema.
· *RegistroPieza*: Clase encargada la gestión de piezas.

MENÚ, PRUEBA Y FÁBRICA

Se ha diseñado una sencilla estructura de menús para poder comprobar todas las funcionalidades implementadas en esta práctica, para ello en la Clase Menu, se ofrece un método público que accede al menú principal, y desde el que se navega a las distintas opciones, mediante llamadas a métodos privados de esta clase o a métodos públicos de clases externas.

Por otro lado, se ha incluido una clase llamada Prueba, que genera:
- Empleados de prueba con nombres y apellidos aleatorios.
- Clientes de prueba con nombres y apellidos aleatorios.
- Muebles al sistema. Instancias de subclases de Mueble.
- Pedidos realizados por clientes instanciados previamente.

El método `cargarDatos` de la clase Prueba es el método encargado de cargar datos iniciales en el sistema.
Por su parte, el método main() de la clase Fabrica inicia la interfaz del sistema y lanza el entorno de ejecución.

ANEXO II.

CÓDIGO FUENTE.

1. Clase Fabrica

```
/**
 * Clase principal del proyecto para la práctica de la asignatura Programación
 * Orientada a Objetos, para el curso 2019/2020
 *
 * @author Arturo Barba Rodríguez abarba54@alumno.uned.es
 * @version 1.0
 */
public class Fabrica
{
   public static void main(String[] args) {
      Prueba.cargarDatos();
      Menu.iniciar();
   }
}
```

2. Clase Menu

```
import java.util.*;

import modelo.persona.empleado.*;
import modelo.persona.cliente.*;
import modelo.pedido.Pedido;
import modelo.mueble.Mueble;
import modelo.mueble.mesa.*;
import modelo.mueble.silla.*;
import modelo.enumerable.*;
import servicio.RegistroEmpleado;
import servicio.RegistroMueble;
import servicio.RegistroPedido;
import servicio.RegistroCliente;
import servicio.Entrada;
import servicio.Salida;

/**
 * Clase encargada de proporcionar el conjunto de menus y submenus principales
 *
 * @author Arturo Barba
 */
public class Menu
{
   /**
    * Menú principal. Inicia el menú.
    */
   public static void iniciar() {
```

```
boolean firstTime = true;
do {
   if (!firstTime) {
      Salida.salto();
      Salida.salto();
      Salida.salto();
      Salida.mostrar(".");
      Salida.mostrar(".");
      Salida.mostrar(".");
      Entrada.getString();
      Salida.mostrar("Pulse 0 para salir y cualquier otra letra para continuar");
      Salida.salto();
      Salida.salto();
      Salida.salto();
      String salir = Entrada.getString();
      if (salir.length() == 1 && salir.charAt(0) == '0'){
         Salida.mostrar("Cerrando aplicación");
         return;
      }
   } else {
      firstTime = false;
   }

   mostrarPrincipal();
   int op = Entrada.getInt(0,14);
   switch (op) {
      case 0:
      Salida.mostrar("Vuelva pronto");
      return;
      case 1:
      gestionArtesanos();
      break;
      case 2:
      RegistroEmpleado.reponerPiezas();
      break;
      case 3:
      RegistroPedido.anotarPedido();
      break;
      case 4:
      RegistroCliente.verPresupuesto();
      break;
      case 5:
      RegistroPedido.finalizarPedido();
      break;
      case 6:
      RegistroEmpleado.asignarPresupuesto();
      break;
      case 7:
```

```
                RegistroEmpleado.notificarClienteEstado();
                break;
                case 8:
                RegistroEmpleado.verPedidosArtesano();
                break;
                case 9:
                RegistroEmpleado.terminarPedidoArtesano();
                break;
                case 10:
                RegistroEmpleado.pararPedido();
                break;
                case 11:
                gestionClientes();
                break;
                case 12:
                gestionEmpleados();
                break;
                case 13:
                gestionMuebles();
                break;
                case 14:
                gestionPedidos();
                break;
            }
        } while (true);
    }

    /**
     * Submenú de gestión de pedidos, que llama a la clase encargada de
     * obtener, modificar, insertar o eliminar datos y ofrece diferentes
     * opciones a realizar.
     */
    private static void gestionPedidos() {
        Salida.mostrar("[1] Ver el listado completo de pedidos");
        Salida.mostrar("[2] Ver los pedidos en un estado determinado");
        Salida.mostrar("[3] Buscar un pedido");
        Salida.mostrar("[4] Buscar un pedido en un estado determinado");
        Salida.mostrar("[0] Menu principal");
        int op = Entrada.getInt(0, 4);
        switch (op) {
            case 0:
            return;
            case 1:
            RegistroPedido.listarPedidos();
            break;
            case 2:
            RegistroPedido.listarPedidosEstado();
            break;
```

```
            case 3:
            RegistroPedido.buscarPedido();
            break;
            case 4:
            RegistroPedido.menuBuscarPedido();
            break;
        }
    }


    /**
     * Submenú de gestión de artesanos por parte de un jefe.
     * Ofrece diferentes opciones a realizar, entre ellas ver el progreso
     * del trabajo de un artesano o asignar trabajo a este.
     */
    private static void gestionArtesanos() {
        Salida.mostrar("[1] Asignar trabajo a un artesano");
        Salida.mostrar("[2] Ver el progreso de un artesano");
        Salida.mostrar("[0] Menu principal");
        int op = Entrada.getInt(0, 2);
        switch (op) {
            case 0:
            return;
            case 1:
            RegistroEmpleado.asignarTrabajo();
            break;
            case 2:
            RegistroEmpleado.verPedidosArtesano();
            break;
        }
    }

    /**
     * Submenú de gestión de empleados, que llama a la clase encargada de
     * obtener, modificar, insertar o eliminar datos y ofrece diferentes
     * opciones a realizar.
     */
    private static void gestionEmpleados() {
        Salida.mostrar("[1] Alta empleado");
        Salida.mostrar("[2] Eliminar empleado");
        Salida.mostrar("[3] Editar empleado");
        Salida.mostrar("[0] Menu principal");
        int op = Entrada.getInt(0, 3);
        switch (op) {
            case 0:
            return;
            case 1:
            RegistroEmpleado.altaEmpleado();
```

```
            break;
            case 2:
            RegistroEmpleado.eliminarEmpleado();
            break;
            case 3:
            RegistroEmpleado.editarEmpleado();
            break;
        }
    }

    /**
     * Submenú de gestión de clientes, que llama a la clase encargada de
     * obtener, modificar, insertar o eliminar datos y ofrece diferentes
     * opciones a realizar.
     */
    private static void gestionClientes() {
        Salida.mostrar("[1] Alta cliente");
        Salida.mostrar("[2] Eliminar cliente");
        Salida.mostrar("[3] Editar cliente");
        Salida.mostrar("[0] Menu principal");
        int op = Entrada.getInt(0, 3);
        switch (op) {
            case 0:
            return;
            case 1:
            RegistroCliente.altaCliente();
            break;
            case 2:
            RegistroCliente.eliminarCliente();
            break;
            case 3:
            RegistroCliente.editarCliente();
            break;
        }
    }

    /**
     * Submenú de gestión de muebles, que llama a la clase encargada de
     * obtener, modificar, insertar o eliminar datos y ofrece diferentes
     * opciones a realizar.
     */
    private static void gestionMuebles() {
        Salida.mostrar("[1] Añadir un mueble al sistema");
        Salida.mostrar("[2] Eliminar un mueble del sistema");
        Salida.mostrar("[3] Listar muebles");
        Salida.mostrar("[0] Menu principal");
        int op = Entrada.getInt(0, 3);
        switch (op) {
```

```
            case 0:
            return;
            case 1:
            RegistroMueble.addMuebles();
            break;
            case 2:
            RegistroMueble.eliminarMueble();
            break;
            case 3:
            RegistroMueble.listaMuebles();
            break;
        }
    }

    /**
     * Muestra las opciones del menú principal.
     */
    private static void mostrarPrincipal() {
        Salida.mostrar("MENU PRINCIPAL");
        Salida.separadorGordo();
        Salida.salto();
        Salida.mostrar("ACCIONES DE JEFE");
        Salida.salto();
        Salida.separadorFino();
        Salida.mostrar("[1] Gestionar trabajo de artesanos");
        Salida.mostrar("[2] Reponer stock de piezas");
        Salida.separadorFino();
        Salida.salto();
        Salida.mostrar("ACCIONES DE CLIENTE");
        Salida.separadorFino();
        Salida.mostrar("[3] Realizar nuevo pedido");
        Salida.mostrar("[4] Aceptar/Rechazar presupuestos");
        Salida.mostrar("[5] Recoger o confirmar recepción de pedidos");
        Salida.separadorFino();
        Salida.salto();
        Salida.mostrar("ACCIONES DE COMERCIAL");
        Salida.mostrar("[6] Asignar presupuesto a un pedido");
        Salida.mostrar("[7] Notificar a cliente pendiente recogida");
        Salida.separadorFino();
        Salida.salto();
        Salida.mostrar("ACCIONES DE ARTESANO");
        Salida.separadorFino();
        Salida.mostrar("[8] Ver pedidos asignados");
        Salida.mostrar("[9] Notificar terminación de fabricación");
        Salida.mostrar("[10] Notificar incidencia");
        Salida.separadorFino();
        Salida.salto();
        Salida.mostrar("GESTIÓN GENERAL");
```

```
        Salida.separadorFino();
        Salida.mostrar("[11] Gestión de clientes");
        Salida.mostrar("[12] Gestión de empleados");
        Salida.mostrar("[13] Gestión de muebles");
        Salida.mostrar("[14] Gestión de pedidos");
        Salida.separadorFino();
        Salida.mostrar("[0] Salir");
        Salida.mostrar("Por favor seleccione una opción:");
    }
}
```

3. Clase Prueba

```
import java.util.*;

import modelo.persona.empleado.*;
import modelo.persona.cliente.*;
import modelo.pedido.Pedido;
import modelo.mueble.Mueble;
import modelo.mueble.mesa.*;
import modelo.mueble.silla.*;
import modelo.enumerable.*;
import servicio.RegistroEmpleado;
import servicio.RegistroMueble;
import servicio.RegistroPedido;
import servicio.RegistroCliente;
import servicio.Entrada;
import servicio.Salida;

/**
 * Clase encargada de generar datos de pruebas. Añade empleados, clientes,
 * muebles y pedidos al sistema.
 *
 * @author Arturo Barba
 */
public class Prueba
{
    /**
     * Variable con el listado de nombres de pruebas.
     */
    private static List<String> nombresPropios;
    /**
     * Variable con el listado de apellidos de pruebas.
     */
    private static List<String> apellidos;

    static {
```

```
        nombresPropios = Arrays.asList(
            "Ángel", "María", "Carmen", "Arturo", "Lourdes", "Carlos",
            "Antonio", "Germán", "Soledad", "Marcos", "Adrián", "Mercedes",
            "Rocío", "Adán", "Luis", "Almudena"
        );

        apellidos = Arrays.asList(
            "García", "Barba", "López", "Olmedo", "Pérez", "Rodríguez",
            "Román", "Mellado", "Escobar", "Banderas", "Delgado", "Ramos",
            "González", "Fernández", "Gómez", "Martín", "Ruiz",
            "Hernández", "Díaz", "Torres", "Muñoz", "Jiménez",
            "Vázquez", "Serrano", "Molina"
        );
    }

    /**
     * Método encargado de cargar datos iniciales en el sistema.
     * Llama a métodos auxiliares para obtener e insertar la información.
     */
    public static void cargarDatos() {
        Salida.mostrar("Generando datos de pruebas...");
        datosEmpleados();
        datosClientes();
        datosMuebles();
        datosPedidos();
        actualizarUnPedido();
        Salida.mostrar("Se han añadido datos al sistema para pruebas");
    }

    /**
     * Método que genera un número entero aleatorio comprendido en un rango.
     * @param min Cota inferior del intervalo
     * @param max Cota superior del intervalo
     * @return Un entero comprendido en el intervalo
     */
    public static int enteroAleatorioEntreRango(int min, int max) {
        return (int) decimalAleatorioEntreRango(min, max);
    }

    /**
     * Método que genera un número float aleatorio comprendido en un rango.
     * @param min Cota inferior del intervalo
     * @param max Cota superior del intervalo
     * @return Un número float comprendido en el intervalo
     */
    public static float decimalAleatorioEntreRango(float min, float max) {
        return (float) (Math.random() * ((max + 1) - min)) + min;
    }
```

```
    /**
     * Método que genera nombre aleatorio, obteniendolo de la variable
     * List<String> nombresPropios.
     * @return El nombre
     */
    public static String generarNombrePropio() {
        return nombresPropios.get(enteroAleatorioEntreRango(0, nombresPropios.size() - 1));
    }

    /**
     * Método que genera un apellido aleatorio, obteniendolo de la variable
     * List<String> apellidos.
     * @return El apellido
     */
    public static String generarApellido() {
        return apellidos.get(enteroAleatorioEntreRango(0, nombresPropios.size() - 1));
    }

    /**
     * Método que genera nombre aleatorio seguido de un primer y un
     * segundo apellido.
     * @return El nombre completo seguido de dos apellidos
     */
    public static String generarNombrePropioCompleto() {
        String nombre = generarNombrePropio();
        String primerApellido = generarApellido();
        String segundoApellido = generarApellido();

        return String.format("%s %s %s", nombre, primerApellido, segundoApellido);
    }

    /**
     * Método que inserta empleados de prueba en el sistema
     */
    private static void datosEmpleados() {
        RegistroEmpleado.addEmpleado(new Jefe(generarNombrePropioCompleto()));
        RegistroEmpleado.addEmpleado(new Comercial(generarNombrePropioCompleto()));
        RegistroEmpleado.addEmpleado(new ArtesanoHora(generarNombrePropioCompleto()));
        RegistroEmpleado.addEmpleado(new ArtesanoPlantilla(generarNombrePropioCompleto()));
    }

    /**
     * Método que inserta clientes de prueba en el sistema
     */
    private static void datosClientes() {
        RegistroCliente.addCliente(new Empresa(generarNombrePropioCompleto()));
        RegistroCliente.addCliente(new Particular(generarNombrePropioCompleto()));
```

```
        RegistroCliente.addCliente(new Particular(generarNombrePropioCompleto()));
    }

    /**
     * Método que inserta muebles de prueba en el sistema
     */
    private static void datosMuebles() {
        RegistroMueble.addMueble(new MesaDeComedor());
        RegistroMueble.addMueble(new MesaCafeMadera());
        RegistroMueble.addMueble(new MesaCafeCristal());
        RegistroMueble.addMueble(new SillaPlegable());
        RegistroMueble.addMueble(new SillaConRuedas());
        RegistroMueble.addMueble(new SillaSinRuedas());
    }

    /**
     * Método que inserta pedidos de prueba en el sistema
     */
    private static void datosPedidos() {
        ArrayList<Mueble> muebles = new ArrayList<Mueble>();
        muebles.add(new MesaDeComedor());
        RegistroPedido.addPedido(new Pedido(muebles, RegistroCliente.getCliente(1)));
        muebles.add(new SillaCocina());
        RegistroPedido.addPedido(new Pedido(muebles, RegistroCliente.getCliente(2)));
        muebles.add(new SillaConRuedas());
        RegistroPedido.addPedido(new Pedido(muebles, RegistroCliente.getCliente(2)));
    }

    /**
     * Método que actualiza el estado de un pedido de prueba en el sistema
     */
    private static void actualizarUnPedido() {
        Comercial c = (Comercial)RegistroEmpleado.getEmpleado(2, "Comercial");
        Pedido p = RegistroPedido.getPedido(1);
        Cliente cli = p.getCliente();
        c.establecerPrecio(p);
        cli.aceptarPresupuesto(p);

    }
}
```

4. Clase Estado

```
package modelo.enumerable;


/**
 * Enumeration class Estado - Listado de todos los estados disponibles
```

```
 * en los que se puede encontrar un pedido
 *
 * @author Arturo Barba
 */
public enum Estado
{
    /**
     * Estado por defecto al crear un pedido y pendiente de asignar presupuesto
     */
    NUEVO {
        public String toString() {
            return "nuevo";
        }
    },
    /**
     * Pendiente de aprobar el presupuesto
     */
    PENDIENTE {
        public String toString() {
            return "pendiente de estudio del presupuesto";
        }
    },
    /**
     * Pedido cancelado (por rechazo de presupuesto por ejemplo)
     */
    CANCELADO {
        public String toString() {
            return "cancelado";
        }
    },
    /**
     * Presupuesto aceptado, pendiente de asignacion de un jefe a un artesano
     */
    ACEPTADO {
        public String toString() {
            return "presupuesto aceptado";
        }
    },
    /**
     * En fabricacion por un artesano
     */
    COLA_FABRICACION {
        public String toString() {
            return "pendiente de fabricación por el artesano";
        }
    },
    /**
     * En fabricacion por un artesano
```

```
     */
    FABRICACION {
        public String toString() {
            return "en fabricación";
        }
    },
    /**
     * Fabricado por un artesano, pendiente de que un comercial notifique al
     * cliente
     */
    FABRICADO {
        public String toString() {
            return "fabricado";
        }
    },
    /**
     * Pendiente de recogida por parte de un cliente
     */
    RECOGER {
        public String toString() {
            return "pendiente de recoger por el cliente";
        }
    },
    /**
     * Pendiente de entrega y confirmación de recepcion por un cliente
     */
    ENTREGAR {
        public String toString() {
            return "pendiente de entrega";
        }
    },
    /**
     * Pedido completado y en manos del cliente
     */
    FINALIZADO {
        public String toString() {
            return "finalizado";
        }
    },
    /**
     * Parado por falta de piezas u otros. Pendiente de reanudación
     */
    PARADO {
        public String toString() {
            return "parado";
        }
    }
}
```

5. Clase Operacion

```
package modelo.enumerable;

/**
 * Enumeration class Operacion - Opreaciones CRUD disponibles
 * Crear, editar o eliminar
 *
 * @author Arturo Barba
 */
public enum Operacion
{
   CREAR, EDITAR, ELIMINAR
}
```

6. Clase Pieza

```
package modelo.enumerable;


/**
 * Enumeration class Pieza - Piezas que forman la fabricación de un mueble
 *
 * @author Arturo Barba
 */
public enum Pieza
{
   TORNILLO, TECHO, FONDO, BASE, PUERTA, CAJON, ESTANTE, OTRO
}
```

7. Clase Mueble

```
package modelo.mueble;



/**
 * Abstract class Mueble - Clase que instancia un mueble del sistema
 *
 * @author: Arturo Barba
 */
public abstract class Mueble
```

```
{
    /**
     * Precio del mueble
     */
    private double precio;
    /**
     * Variable estática encargada de almacenar un id único y autoincremetado
     */
    private static int idActual = 1;
    /**
     * ID del mueble
     */
    private int id;

    /**
     * Constructor de Mueble
     */
    public Mueble() {
        this.id = this.idActual;
        this.idActual++;
    }

    /**
     * Constructor de Mueble que recibe un precio
     * @param precio El precio
     */
    public Mueble(double precio) {
        this.id = this.idActual;
        this.idActual++;
        setPrecio(precio);
    }

    /**
     * Getter del precio
     * @return double El precio
     */
    public double getPrecio() {
        return this.precio;
    }

    /**
     * Setter del precio
     * @param precio El precio
     */
    public void setPrecio(double precio) {
        if (precio > 0)
            this.precio = precio;
    }
```

```
    /**
     * Getter del ID
     * @return int El ID
     */
    public int getId() {
        return this.id;
    }

    /**
     * Método toString
     * @return String Información
     */
    public String toString() {
        return "ID: "+ getId() +". Precio: "+getPrecio()+" euros.";
    }
}
```

8. Clase Mesa

```
package modelo.mueble.mesa;

import modelo.mueble.Mueble;

/**
 * Sublcase de Mueble que engloba a todos los muebles de tipo mesa
 * @author Arturo Barba
 */
public abstract class Mesa extends Mueble
{
    /**
     * Constructor de Mesa
     */
    public Mesa()
    {
        super();
    }

    /**
     * Constructor de Mesa que recibe un precio
     * @param precio El precio
     */
    public Mesa(double precio)
    {
        super(precio);
    }
```

```
    /**
     * Método toString
     * @return String Información
     */
    public String toString() {
        return "Mueble tipo mesa. "+super.toString();
    }
}
```

9. Clase MesaDeCafe

```
package modelo.mueble.mesa;

import modelo.mueble.Mueble;
/**
 * Subclase de Mesa. Clase Abstracta
 *
 * @author Arturo Barba
 */
public abstract class MesaDeCafe extends Mesa
{
    /**
     * Constructor MesaDeCafe
     */
    public MesaDeCafe()
    {
        super();
    }

    /**
     * Constructor MesaDeCafe que recibe un precio
     * @param precio El precio
     */
    public MesaDeCafe(double precio)
    {
        super(precio);
    }

    /**
     * Método toString
     * @return String Información
     */
    public String toString() {
        return super.toString() + " Destinada para tomar café.";
    }
}
```

10. Clase MesaCafeCristal

```
package modelo.mueble.mesa;

import modelo.mueble.Mueble;
/**
 * Subclase de MesaDeCafe. Mesa de café de cristal
 *
 * @author Arturo Barba
 */
public class MesaCafeCristal extends MesaDeCafe
{
    /**
     * Constructor MesaCafeCristal que asigna un precio por defecto
     */
    public MesaCafeCristal()
    {
        super(69.95);
    }

    /**
     * Constructor MesaCafeCristal que recibe un precio
     * @param precio El precio
     */
    public MesaCafeCristal(double precio)
    {
        super(precio);
    }

    /**
     * Método toString
     * @return String Información
     */
    public String toString() {
        return super.toString() + " Fabricada con cristal.";
    }
}
```

11. Clase MesaCafeMadera.

```
package modelo.mueble.mesa;

import modelo.mueble.Mueble;
/**
 * Subclase de MesaDeCafe. Mesa de café de madera
 *
```

```
 * @author Arturo Barba
 */
public class MesaCafeMadera extends MesaDeCafe
{
    /**
     * Constructor MesaCafeMadera que asigna un precio por defecto
     */
    public MesaCafeMadera()
    {
        super(59.95);
    }

    /**
     * Constructor MesaCafeMadera que recibe un precio
     * @param precio El precio
     */
    public MesaCafeMadera(double precio)
    {
        super(precio);
    }

    /**
     * Método toString
     * @return String Información
     */
    public String toString() {
        return super.toString() + " Fabricada de madera.";
    }
}
```

12. Clase MesaDeComedor

```
package modelo.mueble.mesa;

import modelo.mueble.Mueble;

/**
 * Subclase de Mesa
 *
 * @author Arturo Barba
 */
public class MesaDeComedor extends Mesa
{
    /**
     * Constructor MesaDeComedor que asigna un precio por defecto
     */
    public MesaDeComedor()
    {
```

```
        super(199.95);
    }

    /**
     * Constructor MesaDeComedor que recibe un precio
     * @param precio El precio
     */
    public MesaDeComedor(double precio)
    {
        super(precio);
    }

    /**
     * Método toString
     * @return String Información
     */
    public String toString() {
        return super.toString()+" Mesa de comedor.";
    }
}
```

13. Clase MesaDeDormitorio

```
package modelo.mueble.mesa;

import modelo.mueble.Mueble;
/**
 * Sublcase de Mesa
 * @author Arturo Barba
 */
public class MesaDeDormitorio extends Mesa
{
    /**
     * Constructor de MesaDeDormitorio que asigna un precio por defecto
     */
    public MesaDeDormitorio()
    {
        super(109.95);
    }

    /**
     * Constructor MesaDeDormitorio que recibe un precio
     * @param precio El precio
     */
    public MesaDeDormitorio(double precio)
    {
        super(precio);
    }
```

```
    /**
     * Método toString
     * @return String Información
     */
    public String toString() {
        return super.toString()+" Para interiores, especialmente dormitorios.";
    }
}
```

14. Clase Silla

```
package modelo.mueble.silla;

import modelo.mueble.Mueble;

/**
 * Subclase de Mueble que engloba a todas las sillas
 *
 * @author Arturo Barba
 */
public abstract class Silla extends Mueble
{
    /**
     * Constructor Silla
     */
    public Silla()
    {
        super();
    }

    /**
     * Constructor Silla que recibe un precio
     * @param precio el precio
     */
    public Silla(double precio)
    {
        super(precio);
    }

    /**
     * Método toString
     * @return String Información
     */
    public String toString() {
        return "Mueble tipo silla. "+super.toString();
    }
```

```
}
```

15. Clase SillaCocina

```
package modelo.mueble.silla;

/**
 * Subclase de Silla. Silla de cocina
 *
 * @author Arturo Barba
 */
public class SillaCocina extends Silla
{
    /**
     * Constructor SillaCocina que asigna un precio por defecto
     */
    public SillaCocina()
    {
        super(39.95);
    }

    /**
     * Constructor SillaCocina que recibe un precio
     * @param precio El precio
     */
    public SillaCocina(double precio)
    {
        super(precio);
    }

    /**
     * Método toString
     * @return String Información
     */
    public String toString() {
        return super.toString()+ "Silla destinada para cocinas.";
    }
}
```

16. Clase SillaOficina

```
package modelo.mueble.silla;

/**
 * Subclase de Silla. Clase Abstracta Silla Oficina
 *
 * @author Arturo Barba
 */
```

```
public abstract class SillaOficina extends Silla
{
    /**
     * Constructor SillaOficina
     */
    public SillaOficina()
    {
        super();
    }

    /**
     * Constructor SillaOficina que recibe un precio
     * @param precio El precio
     */
    public SillaOficina(double precio)
    {
        super(precio);
    }

    /**
     * Método toString
     * @return String Información
     */
    public String toString() {
        return super.toString()+ "Silla destinada para oficinas.";
    }
}
```

17. Clase SillaConRuedas

```
package modelo.mueble.silla;

/**
 * Subclase de SillaOficina. Silla con ruedas.
 *
 * @author Arturo Barba
 */
public class SillaConRuedas extends SillaOficina
{
    /**
     * Constructor SillaConRuedas que asigna un precio por defecto
     */
    public SillaConRuedas()
    {
        super(99.95);
    }

    /**
```

```
    * Constructor SillaConRuedas que recibe un precio
    * @param precio El precio
    */
   public SillaConRuedas(double precio)
   {
      super(precio);
   }

   /**
    * Método toString
    * @return String Información
    */
   public String toString() {
      return super.toString()+ "Incorpora ruedas.";
   }
}
```

18. Clase SillaSinRuedas

```
package modelo.mueble.silla;

/**
 * Subclase de SillaOficina. Silla sin ruedas
 *
 * @author Arturo Barba
 */
public class SillaSinRuedas extends SillaOficina
{
   /**
    * Constructor SillaSinRuedas que asigna un precio por defecto
    */
   public SillaSinRuedas()
   {
      super(95.95);
   }

   /**
    * Constructor SillaSinRuedas que recibe un precio
    * @param precio El precio
    */
   public SillaSinRuedas(double precio)
   {
      super(precio);
   }

   /**
    * Método toString
```

```
     * @return String Información
     */
    public String toString() {
        return super.toString()+ "No incorpora ruedas.";
    }
}
```

19. Clase SillaPlegable

```
package modelo.mueble.silla;

/**
 * Subclase de Silla. Silla plegable
 *
 * @author Arturo Barba
 */
public class SillaPlegable extends Silla
{
    /**
     * Constructor SillaPlegable que asigna un precio por defecto
     */
    public SillaPlegable()
    {
        super(35.95);
    }

    /**
     * Constructor SillaPlegable que recibe un precio
     * @param precio El precio
     */
    public SillaPlegable(double precio)
    {
        super(precio);
    }

    /**
     * Método toString
     * @return String Información
     */
    public String toString() {
        return super.toString()+ "Silla plegable.";
    }
}
```

20. Clase Pedido

```
package modelo.pedido;

import modelo.persona.empleado.*;
import modelo.persona.cliente.Cliente;
import modelo.mueble.Mueble;
import modelo.enumerable.*;

import java.util.*;

/**
 * Clase Pedido. Clase encarga de instanciar un nuevo pedido realizado
 * por un un cliente que figure en el sistema, fabricado por un artesano,
 * con un precio asignado por un comercial y con productos de tipo Mueble
 *
 * @author Arturo Barba
 */
public class Pedido
{
    /**
     * Variable estática encargada de almacenar un id único y autoincremetado
     */
    private static int idActual = 1;
    /**
     * El ID del pedido
     */
    private int id;
    /**
     * Cliente que ha realizado el pedido
     */
    private Cliente cliente;
    /**
     * Comercial encargado de la gestión del pedido
     */
    private Comercial comercial;
    /**
     * Artesano encargado de la fabricación de los muebles del pedido siempre
     * que proceda, es decir, que el cliente haya aceptado el presupuesto
     * dado por el comercial
     */
    private Artesano artesano;
    /**
     * Listado de todos los muebles que forman parte del pedido
     */
    private ArrayList <Mueble> muebles;
    /**
     * Estado del pedido.
     */
    private Estado estado;
```

```
/**
 * Precio del pedido
 */
private double precio = 0;
/**
 * Variable que indica si el precio ha sido asignado por el comercial
 * correspondiente. Si aún no ha sido asginado se reflajará en el estado
 * y el precio será el de por defecto
 */
private boolean precioAsignado = false;
/**
 * Listado de piezas necesarias para fabricar un pedido.
 * Sera nulo si el estado no es Estado.PARADO
 */
private ArrayList <Pieza> piezasPendientes;

/**
 * Constructor Pedido. Crea un pedido con el cliente y los muebles
 * recibidos, asigna un ID al pedido y le asigna el estado Estado.NUEVO
 * @param muebles Los muebles del pedido
 * @param cliente El cliente que ha realizado el pedido
 */
public Pedido(ArrayList <Mueble> muebles, Cliente cliente)
{
   this.muebles = new ArrayList <Mueble>(muebles);
   this.id = this.idActual;
   this.idActual++;
   this.estado = Estado.NUEVO;
   this.cliente = cliente;
   this.piezasPendientes = new ArrayList <Pieza>();
}

/**
 * Getter del ID
 * @return int ID del pedido
 */
public int getId() {
   return this.id;
}

/**
 * Getter del precio. Devuelve 0 si el precio aun no ha sido asignado
 * por un comercial.
 * @return double precio
 */
public double getPrecio() {
   if (this.precioAsignado) return this.precio;
   return 0;
```

```
    }

    /**
     * Getter de los muebles del pedido
     * @return ArrayList<Mueble> Los muebles
     */
    public ArrayList<Mueble> getMuebles() {
        return this.muebles;
    }

    /**
     * Getter del artesano
     * @return Artesano El artesano
     */
    public Artesano getArtesano() {
        return this.artesano;
    }

    /**
     * Getter del cliente
     * @return Cliente El cliente
     */
    public Cliente getCliente() {
        return this.cliente;
    }

    /**
     * Getter del estado
     * @return Estado El estado del pedido
     */
    public Estado getEstado() {
        return this.estado;
    }

     /**
     * Getter de las piezas necesarias para continuar el pedido.
     * @return ArrayList<Pieza> Las piezas
     */
    public ArrayList<Pieza> getPiezas() {
        return this.piezasPendientes;
    }

    /**
     * Setter estado
     * @param estado El nuevo estado del pedido
     */
    public void setEstado(Estado estado) {
        if (estado!=null)
```

```
            this.estado = estado;
    }

    /**
     * Setter de piezas pendientes
     * @param p Las piezas
     */
    public void setPiezas(ArrayList<Pieza> p) {
        if (p!=null)
            this.piezasPendientes = p;
    }

    /**
     * Setter precio
     * @param precio El nuevo precio del pedido
     */
    public void setPrecio(double v) {
        this.precio = v;
        this.precioAsignado = true;
    }

    /**
     * Setter artesano
     * @param artesano El artesano encargado de la fabricación de los muebles
     * del pedido
     */
    public void setArtesano(Artesano artesano) {
        if (artesano!=null)
            this.artesano = artesano;
    }

    /**
     * Método toString
     * @return String Información del pedido
     */
    public String toString() {
        if (this.precioAsignado)
        return "Pedido con ID: "+ this.id +". "+"Comprado por "
        +this.cliente.toString()+". Precio: "+this.getPrecio()+"€. "+
        "Estado "+getEstado().toString().toLowerCase();
        return "Pedido con ID: "+ this.id +". "+"Comprado por "
        +this.cliente.toString()+
        ". Estado "+getEstado().toString().toLowerCase();
    }
}
```

21. Clase Persona

```
package modelo.persona;

/**
 * Abstract class Persona - Clase que engloba a todos los empleados y clientes
 *
 * @author: Arturo Barba
 */
public abstract class Persona
{
    /**
     * Variable nombre de la clase Persona.
     */
    private String nombre;

    /**
     * Constructor que asigna un nombre por defecto
     */
    public Persona(){
        setNombre("");
    }

    /**
     * Contructor de Persona, que recibe el nombre
     * @param nombre El nombre
     */
    public Persona(String nombre){
        setNombre(nombre);
    }

    /**
     * Getter para devolver el nombre
     * @return nombre
     */
    public String getNombre() {
        return this.nombre;
    }

    /**
     * Setter para asignar o cambiar el nombre
     * @param nombre El nombre
     */
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    /**
     * Método toString encargado de devolver información de la clase Persona
```

```
     * @return String información
     */
    public String toString() {
        return "Nombre: "+getNombre();
    }
}
```

22. Clase Cliente

```
package modelo.persona.cliente;

import modelo.persona.Persona;
import modelo.pedido.Pedido;
import modelo.enumerable.*;

/**
 * Abstract class Cliente - Subclase de Persona con rol de Cliente
 *
 * @author: Arturo Barba
 */
public abstract class Cliente extends Persona
{
    /**
     * Variable estática encargada de almacenar un id único y autoincremetado
     */
    private static int idActual = 1;
    /**
     * ID del cliente
     */
    private int clienteId;

    /**
     * Constructor de Cliente
     */
    public Cliente() {
        super();
        this.clienteId = this.idActual;
        this.idActual++;
    }

    /**
     * Constructor de Cliente que recibe un nombre
     * @param nombre El nombre
     */
    public Cliente(String nombre) {
        super(nombre);
        this.clienteId = this.idActual;
```

```
        this.idActual++;
    }

    /**
     * Getter del ID del cliente
     * @return int El ID
     */
    public int getClienteId() {
        return this.clienteId;
    }

    /**
     * Método encargado de rechazar el presupuesto de un pedido
     * @param Pedido El pedido
     */
    public void rechazarPresupuesto(Pedido p){
        p.setEstado(Estado.CANCELADO);
    }

    /**
     * Método encargado de aceptar el presupuesto de un pedido
     * @param Pedido El pedido
     */
    public void aceptarPresupuesto(Pedido p){
        p.setEstado(Estado.ACEPTADO);
    }

    /**
     * Método toString
     * @return String Información
     */
    public String toString() {
        return super.toString() + " id: "+getClienteId();
    }
}
```

23. Clase Empresa

```
package modelo.persona.cliente;

import modelo.persona.Persona;
/**
 * Subclase Cliente - Cliente a nombre de una empresa
 *
 * @author Arturo Barba
 */
public class Empresa extends Cliente
```

```
{
    /**
     * Constructor de Empresa
     */
    public Empresa() {
        super();
    }

    /**
     * Constructor de Empresa que recibe el nombre
     * @param nombre El nombre de la empresa
     */
    public Empresa(String nombre) {
        super(nombre);
    }

    /**
     * Método toString
     * @return String Información
     */
    public String toString() {
        return super.toString() + ". Tipo cliente: empresa";
    }
}
```

24. Clase Particular

```
package modelo.persona.cliente;

import modelo.persona.Persona;
/**
 * Subclase Cliente - Cliente particular
 *
 * @author Arturo Barba
 */
public class Particular extends Cliente
{
    /**
     * Constructor de Particular
     */
    public Particular() {
        super();
    }

    /**
     * Constructor de Particular que recibe un nombre
     * @param nombre El nombre del empresario
```

```
     */
    public Particular(String nombre) {
       super(nombre);
    }

    /**
     * Método toString
     * @return String Información
     */
    public String toString() {
        return super.toString() + ". Tipo cliente: particular";
    }
}
```

25. Clase Empleado

```
package modelo.persona.empleado;

import modelo.persona.Persona;


/**
 * Abstract class Empleado - Clase que engloba a todos los empleados y hereda
 * de Persona
 *
 * @author: Arturo Barba
 */
public abstract class Empleado extends Persona
{
    /**
     * Variable estática encargada de almacenar un id único y autoincremetado
     */
    private static int idActual = 1;
    /**
     * El id del empleado
     */
    private int empleadoId;

    /**
     * Constructor de Empleado
     */
    public Empleado() {
       super();
       this.empleadoId = this.idActual;
       this.idActual++;
    }
```

```
    /**
     * Constructor de Empleado que recibe un nombre
     * @param nombre El nombre
     */
    public Empleado(String nombre) {
        super(nombre);
        this.empleadoId = this.idActual;
        this.idActual++;
    }

    /**
     * Getter del ID del empleado
     * @return int El ID
     */
    public int getEmpleadoId() {
        return this.empleadoId;
    }

    /**
     * Método toString
     * @return String Información del empleado
     */
    public String toString() {
        return super.toString() + " id: "+getEmpleadoId();
    }
}
```

26. Clase Comercial

```
package modelo.persona.empleado;

import modelo.persona.Persona;
import modelo.pedido.Pedido;
import modelo.enumerable.Estado;
import modelo.persona.empleado.*;
import modelo.persona.cliente.Cliente;
import modelo.mueble.Mueble;
import modelo.enumerable.Estado;

import java.util.*;

/**
 * Clase Comercial - Subclase de Empleado que especifica un empleado que
 * trabaja como comercial
 *
 * @author Arturo Barba
```

```
 */
public class Comercial extends Empleado
{
    /**
     * Constructor de subclase Comercial
     */
    public Comercial() {
        super();
    }

    /**
     * Constructor de subclase Comercial que recibe el nombre
     * @param nombre El nombre
     */
    public Comercial(String nombre) {
        super(nombre);
    }

    /**
     * Método que actualiza el estado de un Pedido a Estado.RECOGER
     * @param Pedido El pedido
     */
    public void notificarClienteRecoger(Pedido p) {
        p.setEstado(Estado.RECOGER);
    }

    /**
     * Método que actualiza el estado de un Pedido a Estado.ENTREGAR
     * @param Pedido El pedido
     */
    public void notificarClienteEntregar(Pedido p) {
        p.setEstado(Estado.ENTREGAR);
    }

    /**
     * Método que establece un precio a un Pedido
     * @param Pedido El pedido
     */
    public void establecerPrecio(Pedido p) {
        double total = 0;
        ArrayList<Mueble> m = p.getMuebles();
        for (int i = 0; i < m.size(); i++) {
            total += m.get(i).getPrecio();
        }
        double scale = Math.pow(10, 2);

        total = Math.round(total * scale) / scale;
        p.setPrecio(total);
```

```
        p.setEstado(Estado.PENDIENTE);
    }

    /**
     * Método toString
     * @return String Información
     */
    public String toString() {
        return super.toString() + ". Puesto: comercial";
    }
}
```

27. Clase Jefe

```
package modelo.persona.empleado;

import java.util.*;
import modelo.persona.Persona;
import modelo.pedido.*;
import modelo.enumerable.*;

/**
 * Clase Jefe - Subclase de Empleado con el rol de Jefe
 *
 * @author Arturo Barba
 */
public class Jefe extends Empleado
{
    /**
     * Constructor de Jefe
     */
    public Jefe() {
        super();
    }

    /**
     * Constructor de Jefe que recibe el nombre
     * @param nombre El nombre
     */
    public Jefe(String nombre) {
        super(nombre);
    }

    /**
     * Método que asigna un pedido a un Artesano para que comience a trabajar
     * en él.
     * @param Pedido El pedido
```

```
     * @param Artesano El empleado artesano
     */
    public void asignarPedido(Pedido pedido, Artesano artesano)
    {
        pedido.setEstado(Estado.COLA_FABRICACION);
        pedido.setArtesano(artesano);
        artesano.asignarTrabajo(pedido);
    }

    /**
     * Método que repone las piezas de un pedido y lo reasigna a su
     * Artesano para que comience a trabajar en él.
     * @param Pedido El pedido
     */
    public void reponerPiezas(Pedido pedido)
    {
        pedido.setEstado(Estado.COLA_FABRICACION);
        pedido.setPiezas(new ArrayList<Pieza>());
    }

    /**
     * Método toString
     * @return String información
     */
    public String toString() {
        return super.toString() + ". Puesto: jefe";
    }
}
```

28. Clase Artesano

```
package modelo.persona.empleado;


import modelo.persona.Persona;
import modelo.pedido.Pedido;
import modelo.enumerable.*;
import servicio.*;

import java.util.*;

/**
 * Abstract class Artesano - Subclase de Empleado
 *
 * @author: Arturo Barba
 */
public abstract class Artesano extends Empleado
```

```
{
   /**
    * Listado de pedidos en los que trabaja o ha trabajado el Artesano
    */
   private List<Pedido> pedidos;

   /**
    * Constructor de Artesano
    */
   public Artesano() {
      super();
      this.pedidos = new ArrayList <Pedido>();
   }

   /**
    * Constructor de Artesano que recibe el nombre
    * @param nombre El nombre
    */
   public Artesano(String nombre) {
      super(nombre);
      this.pedidos = new ArrayList <Pedido>();
   }

   /**
    * Getter de pedidos
    * @return List<Pedido> Los pedidos
    */
   public List<Pedido> getPedidos() {
       return this.pedidos;
   }

   /**
    * Lista todos los pedidos del artesano en un estado
    * @param e El estado de los pedidos a obtener
    * @return List<Pedido> Los pedidos
    */
   public List<Pedido> getPedidos(Estado e) {
       return RegistroPedido.getPedidos(e, this);
   }

   /**
    * Lista todos los pedidos en fabricación por el artesano
    */
   public void verPedidosPendientes() {
       RegistroPedido.listarPedidos(Estado.COLA_FABRICACION, this);
   }

   /**
```

```
     * Método que asigna un Pedido a un artesano
     * @param Pedido El pedido
     */
    public void asignarTrabajo(Pedido pedido)
    {
        this.pedidos.add(pedido);
        pedido.setEstado(Estado.COLA_FABRICACION);
        comenzarSiguienteTrabajo();
    }

    /**
     * Método que actualiza el estado de un Pedido cuando el Artesano ha
     * terminado con su fabricación y comienza con la fabricación del
     * siguiente, siempre que haya pendientes
     * @param Pedido El pedido
     */
    public void finalizarTrabajo(Pedido pedido)
    {
        pedido.setEstado(Estado.FABRICADO);
        comenzarSiguienteTrabajo();
    }

    /**
     * Método que actualiza el estado de un Pedido a Estado.PARADO
     * y comienza con la fabricación el siguiente, siempre que haya
     * pendientes
     * @param Pedido El pedido
     */
    public void detenerTrabajo(Pedido p) {
        p.setEstado(Estado.PARADO);
        comenzarSiguienteTrabajo();
    }

    /**
     * Método que actualiza el estado de un Pedido a Estado.PARADO por falta de
     * piezas y comienza con la fabricación el siguiente, siempre que haya
     * pendientes
     * @param pedido El pedido
     * @param piezas El listado con las piezas que faltan
     */
    public void detenerTrabajo(Pedido pedido, ArrayList<Pieza> piezas) {
        pedido.setEstado(Estado.PARADO);
        pedido.setPiezas(piezas);
        comenzarSiguienteTrabajo();
    }

    /**
     * Devuelve el siguiente pedido con el que debe comenzar a trabajar
```

```
     * un artesano, devolverá el pedido que lleve mas tiempo en el sistema,
     * es decir, el que tiene menor ID
     * @return Pedido El pedido
     */
    private Pedido getSiguientePedido() {
        return RegistroPedido.getSiguientePedido(getPedidos(Estado.COLA_FABRICACION));
    }

    /**
     * Consulta en el sistema la siguiente tarea a realizar, y en el caso de
     * que exista una y el artesano no este involucrado en otra tarea
     * se comienza con la fabricación de esta
     */
    private void comenzarSiguienteTrabajo() {
                                    if      (getPedidos(Estado.FABRICACION).size()      ==      0      &&
getPedidos(Estado.COLA_FABRICACION).size() > 0) {
            getSiguientePedido().setEstado(Estado.FABRICACION);
        }
    }
}
```

29 Clase ArtesanoHora

```
package modelo.persona.empleado;

import modelo.persona.Persona;

/**
 * Subclase de Artesano. Artesano que trabaja por horas.
 *
 * @author Arturo Barba
 */
public class ArtesanoHora extends Artesano
{
    /**
     * Constructor de ArtesanoHora
     */
    public ArtesanoHora() {
      super();
    }

    /**
     * Constructor de ArtesanoHora que recibe un nombre
     * @param nombre El nombre
     */
    public ArtesanoHora(String nombre) {
      super(nombre);
```

```
    }

    /**
     * Método toString
     * @return String Información
     */
    public String toString() {
        return super.toString() + ". Puesto: artesano con contrato por hora";
    }
}
```

30. Clase ArtesanoPlantilla

```
package modelo.persona.empleado;

import modelo.persona.Persona;

/**
 * Subclase de Artesano. Artesano que figura en la plantilla
 *
 * @author Arturo Barba
 */
public class ArtesanoPlantilla extends Artesano
{
    /**
     * Constructor de ArtesanoPlantilla
     */
    public ArtesanoPlantilla() {
        super();
    }

    /**
     * Constructor de ArtesanoPlantilla que recibe un nombre
     * @param nombre El nombre
     */
    public ArtesanoPlantilla(String nombre) {
        super(nombre);
    }

    /**
     * Método toString
     * @return String Información
     */
    public String toString() {
        return super.toString() + ". Puesto: artesano en plantilla";
    }
}
```

31. Clase Entrada

```
package servicio;

import java.util.*;

/**
 * Clase encargada de la lectura de datos del sistema
 *
 * @author Arturo Barba
 */
public class Entrada
{
    /**
     * Variable estática scanner encargada de leer datos de entrada
     */
    private static Scanner lector = new Scanner(System.in);

    /**
     * Lee y devuelve un número entero introducido al sistema
     * @return int El número
     */
    public static int getInt() {
        lector.reset();
        return lector.nextInt();
    }

    /**
     * Lee y devuelve un número entero introducido al sistema comprendido en
     * un intervalo
     * @param min La cota inferior del intervalo
     * @param max La cota superior del intervalo
     * @return int El número
     */
     public static int getInt(int min, int max) {
        lector.reset();
        int op;
        do {
            op = lector.nextInt();
        } while(op > max || min > op);
        return op;
    }

    /**
     * Lee y devuelve una cadena introducida en el sistema
     * @return String La cadena
```

```
     */
    public static String getString() {
        lector.reset();
        return lector.nextLine().trim();
    }

    /**
     * Limpia el scanner
     */
    public static void limpiar() {
        lector.next();
    }
}
```

32. Clase Salida

```
package servicio;

import modelo.enumerable.Operacion;

/**
 * Clase encargada de mostrar información al usuario por consola
 *
 * @author Arturo Barba
 */
public class Salida
{
    /**
     * Muestra un separador fino
     */
    public static void separadorFino() {
        mostrar("------------------------------------------------------------");
    }

    /**
     * Muestra un separador gordo
     */
    public static void separadorGordo() {
        mostrar("============================================================");
    }

    /**
     * Muestra un mensaje
     * @param mensaje El mensaje
     */
    public static void mostrar(String mensaje) {
        System.out.println(mensaje);
```

```
    }

    /**
     * Realiza un salto de línea
     */
    public static void salto() {
        System.out.println();
    }

    /**
     * Muestra un submenú de opciones con las Sublclases de Silla
     */
    public static void mostrarMensajeSillas() {
        Salida.mostrar("[1] Sillas de oficina");
        Salida.mostrar("[2] Sillas de cocina");
        Salida.mostrar("[3] Sillas plegable");
        Salida.mostrar("[0] Salir");
    }

    /**
     * Muestra un submenú de opciones con las Sublclases de Mesa
     */
    public static void mostrarMensajeMesas() {
        Salida.mostrar("[1] Mesa de café");
        Salida.mostrar("[2] Mesa de dormitorio");
        Salida.mostrar("[3] Mesa de comedor");
        Salida.mostrar("[0] Salir");
    }

    /**
     * Muestra un submenú de opciones con las Sublclases de SillaDeOficina
     */
    public static void mostrarMensajeSillaOficina() {
        Salida.mostrar("[1] Silla de oficina con ruedas");
        Salida.mostrar("[2] Silla de oficina sin ruedas");
        Salida.mostrar("[0] Salir");
    }

    /**
     * Muestra un submenú de opciones con las Sublclases de MesaDeCafé
     */
    public static void mostrarMensajeMesaCafe() {
        Salida.mostrar("[1] Mesa de cristal");
        Salida.mostrar("[2] Mesa de madera");
        Salida.mostrar("[0] Salir");
    }

    /**
```

```
     * Muestra un submenú de opciones con las Sublclases de Mueble
     */
    public static void mostrarMensajeMuebles() {
        Salida.mostrar("[1] Ver sillas");
        Salida.mostrar("[2] Ver mesas");
        Salida.mostrar("[0] Salir");
    }

    /**
     * Muestra un submenú de opciones con las Sublclases de Cliente
     */
    public static void mostrarMensajeAltaCliente() {
        mostrar("[1] Cliente particular");
        mostrar("[2] Empresa");
        mostrar("[0] Salir");
    }

    /**
     * Muestra un submenú de opciones con las Sublclases de Empleado
     */
    public static void mostrarMensajeAltaEmpleado() {
        mostrar("[1] Jefe");
        mostrar("[2] Comercial");
        mostrar("[3] Artesano");
        mostrar("[0] Salir");
    }

    /**
     * Muestra un submenú de opciones con las Sublclases de Artesano
     */
    public static void mostrarMensajeAltaArtesano() {
        mostrar("[1] Artesano por hora");
        mostrar("[2] Artesano en plantilla");
        mostrar("[0] Salir");
    }

    /**
     * Muestra un submenú para añadir un pedido
     */
    public static void mostrarOpcionesPedido() {
        Salida.mostrar("Introduzca uno de los siguientes valores");
        Salida.mostrar("[1]. Para añadir muebles a su pedido");
        Salida.mostrar("[2]. Para finalizar la compra");
        Salida.mostrar("[0]. Para salir y cancelar el pedido");
    }
}
```

33. Clase Registro Cliente

```
package servicio;


import java.util.*;

import modelo.persona.empleado.*;
import modelo.persona.cliente.*;
import modelo.pedido.Pedido;
import modelo.mueble.Mueble;
import modelo.mueble.mesa.*;
import modelo.mueble.silla.*;
import modelo.enumerable.*;
import servicio.RegistroEmpleado;
import servicio.RegistroMueble;
import servicio.RegistroPedido;
import servicio.Entrada;
import servicio.Salida;

/**
 * Clase encargada de almacenar, insertar, editar y eliminar los clientes que figuran en el sistema.
 *
 * @author Arturo Barba
 */
public class RegistroCliente
{
    /**
     * Variable estática con el listado de todos los clientes
     */
    private static List<Cliente> clientes = new ArrayList<Cliente>();

    /**
     * Getter cliente. Devuelve in cliente del sistema en función de su ID
     * @param id El ID el cliente
     * @return Cliente El cliente
     */
    public static Cliente getCliente(int id) {
        Iterator<Cliente> iter = clientes.iterator();
        boolean encontrado = false;
        Cliente cliente;
        while (iter.hasNext() && !encontrado) {
            cliente = iter.next();
            if (id == cliente.getClienteId()) {
                encontrado = !encontrado;
                return cliente;
            }
        }
```

```
        return null;
    }

    /**
     * Muestra el listado de todos los clientes que figuran en el sistema
     */
    public static void listarClientes() {
        Salida.separadorGordo();
        Salida.mostrar("LISTADO DE CLIENTES");
        Salida.separadorFino();
        if (clientes.size()<=0) {
            Salida.mostrar("No dispone de ningun cliente en la base de datos");
            Salida.separadorGordo();
            return;
        }
        Iterator<Cliente> iter  = clientes.iterator();
        Cliente cliente;

        while (iter.hasNext()) {
            cliente = iter.next();
            Salida.mostrar(cliente.toString());
        }
        Salida.separadorGordo();
    }

    /**
     * Indica si hay clientes en el sistema
     * @return boolean Si existen clientes o no
     */
    public static boolean tieneClientes() {
        return clientes.size() > 0;
    }

    /**
     * Muestra un submenú de selección y permite ver, aceptar o rechazar los presupuestos generados por un comercial a un pedido realizado por
     * el cliente seleccionado
     */
    public static void verPresupuesto(){
        if (!tieneClientes()){
            Salida.mostrar("No existen clientes en el sistema");
            return;
        }

        if (!RegistroPedido.tienePedidos(Estado.PENDIENTE)){
            Salida.mostrar("No existen pedidos en este estado");
            return;
        }
```

```
int id;
Cliente cliente;
do {
   RegistroCliente.listarClientes();
   Salida.mostrar("Introduzca el id del cliente o 0 para salir");
   id = Entrada.getInt();
   if (id == 0) return;
   cliente = getCliente(id);
} while(cliente==null);

if(RegistroPedido.getPedidos(Estado.PENDIENTE, cliente).size() == 0){
   Salida.mostrar("El cliente no pendiente ningún presupuesto por aprobar");
   return;
}

Pedido p;
do {
   RegistroPedido.listarPedidos(Estado.PENDIENTE, cliente);
   Salida.mostrar("Introduzca el id del pedido o 0 para salir");
   id = Entrada.getInt();
   if (id == 0) return;
   p = RegistroPedido.getPedido(id, Estado.PENDIENTE, cliente);
} while(p==null);

Salida.mostrar("Seleccionado el pedido: "+p.toString());

int opcion;
do {
   Salida.mostrar("Seleccione una opcion:");
   Salida.mostrar("[1] Aceptar presupuesto");
   Salida.mostrar("[2] Rechazar presupuesto");
   Salida.mostrar("[0] Salir");
   opcion = Entrada.getInt();
} while (opcion > 2 || opcion < 0);

switch (opcion) {
   case 0:
   return;
   case 1:
   cliente.aceptarPresupuesto(p);
   Salida.mostrar("Presupuesto aceptado. Pendiente de fabricacción");
   break;
   case 2:
   cliente.rechazarPresupuesto(p);
   Salida.mostrar("Presupuesto rechazado. El pedido ha sido cancelado");
   break;
}
```

```
    }

    /**
     * Muestra un submenú y lee entradas del sistema para dar de alta a un nuevo cliente
     */
    public static void altaCliente() {
        int opcion;
        do {
            Salida.mostrarMensajeAltaCliente();
            opcion = Entrada.getInt();
        } while (opcion > 2 || opcion < 0);

        switch (opcion) {
            case 0:
            return;
            case 1:
            addParticular();
            break;
            case 2:
            addEmpresa();
            break;
        }
    }

    /**
     * Añade un cliente al sistema y muestra un mensaje de éxito
     * @param c El cliente
     */
    public static void addCliente(Cliente c) {
        clientes.add(c);
        Salida.mostrar("El cliente: "+c.toString());
        Salida.mostrar("Ha sido añadido al sistema");
    }

    /**
     * Muestra un listado de todos los clientes y pide la seleccion de un usuario por id para eliminarlo
del sistema
     */
    public static void eliminarCliente() {
        listarClientes();
        if (!tieneClientes()) {
            return;
        }
        Salida.mostrar("Introduzca el id del cliente que quiere eliminar");
        deleteCliente(Entrada.getInt());
    }

    /**
```

```
     * Muestra un listado de todos los clientes y pide la selección de un cliente por id para editar sus datos en el sistema
     */
    public static void editarCliente() {
        listarClientes();
        if (!tieneClientes()) {
            return;
        }
        Salida.mostrar("Introduzca el id del cliente que quiere editar");
        int id = Entrada.getInt();
        Cliente cliente = getCliente(id);

        if (cliente==null) {
            Salida.mostrar("No existe el cliente con ID: "+id);
            return;
        }

        Salida.mostrar("Inserte el nuevo nombre del cliente");
        Entrada.getString();
        cliente.setNombre(Entrada.getString());
    }

    /**
     * Elimina a un cliente del sistema. Muestra un mensaje de éxito o de error.
     * @param id El id del cliente
     */
    private static void deleteCliente(int id) {
        Cliente cliente = getCliente(id);
        if (cliente == null) {
            Salida.mostrar("No existe el cliente con ID: "+id);
            return;
        }
        clientes.remove(cliente);
    }

    /**
     * Crea un cliente Particular con el nombre introducido por el usuario y realiza
     * una llamada al método auxiliar encargado de guardar el cliente en el sistema
     */
    private static void addParticular() {
        Salida.mostrar("Introduzca el nombre del cliente particular");
        Entrada.getString();
        Particular e = new Particular(Entrada.getString());
        addCliente(e);
    }

    /**
     * Crea un cliente Empresa con el nombre introducido por el usuario y realiza
```

```
     * una llamada al método auxiliar encargado de guardar el cliente en el sistema
     */
    private static void addEmpresa() {
        Salida.mostrar("Introduzca el nombre de la empresa");
        Entrada.getString();
        Empresa e = new Empresa(Entrada.getString());
        addCliente(e);
    }

}
```

34. Clase RegistroEmpleado

```
package servicio;

import java.util.*;
import modelo.persona.empleado.*;
import modelo.persona.cliente.*;
import modelo.pedido.Pedido;
import modelo.mueble.Mueble;
import modelo.mueble.mesa.*;
import modelo.mueble.silla.*;
import modelo.enumerable.*;
import servicio.*;
import servicio.Salida;

/**
 * Clase encargada de almacenar, insertar, editar y eliminar los empleados que figuran en el sistema.
 *
 * @author Arturo Barba
 */
public class RegistroEmpleado
{
    /**
     * Variable estática con el listado de todos los empleados
     */
    private static List<Empleado> empleados = new ArrayList<Empleado>();

    /**
     * Devuelve un empleado
     * @param id El ID
     * return Empleado El empleado
     */
    public static Empleado getEmpleado(final int id) {
        final Iterator<Empleado> iter  = empleados.iterator();
        Empleado empleado;
        while (iter.hasNext()) {
```

```
            empleado = iter.next();
            if (id == empleado.getEmpleadoId()) {
                return empleado;
            }
        }
        return null;
    }

    /**
     * Devuelve un empleado que figura en un puesto
     * @param id El ID
     * @param cargo El puesto
     * return Empleado El empleado
     */
    public static Empleado getEmpleado(final int id, final String cargo) {
        final Iterator<Empleado> iter  = empleados.iterator();
        Empleado empleado;
        while (iter.hasNext()) {
            empleado = iter.next();
            if (id == empleado.getEmpleadoId()
            && empleado.getClass().getSimpleName().equals(cargo)) {
                return empleado;
            }
        }
        return null;
    }

    /**
     * Getter. Devuelve un artesano
     *
     * @param id El id del empleado
     * @return Artesano El artesano
     */
    public static Artesano getArtesano(final int id) {
        final Iterator<Empleado> iter = empleados.iterator();
        final boolean encontrado = false;
        Empleado e;
        while (iter.hasNext()) {
            e = iter.next();
            if ((e.getClass().getSimpleName().equals("ArtesanoHora")
                    || e.getClass().getSimpleName().equals("ArtesanoPlantilla")) && e.getEmpleadoId() ==
id) {
                return (Artesano) e;
            }
        }
        return null;
    }
```

```
/**
 * Muestra todos los empleados que figuran en el sistema
 */
public static void listarEmpleados() {
   Salida.separadorGordo();
   Salida.mostrar("LISTADO DE EMPLEADOS");
   Salida.separadorFino();
   if (empleados.size()<=0) {
      Salida.mostrar("No dispone de ningun empleado en su plantilla");
      Salida.separadorGordo();
      return;
   }
   final Iterator<Empleado> iter  = empleados.iterator();
   Empleado empleado;

   while (iter.hasNext()) {
      empleado = iter.next();
      Salida.mostrar(empleado.toString());
   }
   Salida.separadorGordo();
}

/**
 * Muestra todos los empleados que figuran en el sistema en un puesto
 * @param cargo El puesto
 */
public static void listarEmpleados(final String cargo) {
   Salida.separadorGordo();
   Salida.mostrar("LISTADO DE "+cargo.toUpperCase());
   Salida.separadorFino();
   if (!tiene(cargo)) {
      Salida.mostrar("No dispone de ningun "+cargo+" en su plantilla");
      Salida.separadorGordo();
      return;
   }
   final Iterator<Empleado> iter  = empleados.iterator();
   final Empleado empleado;

   Empleado puesto;
   while (iter.hasNext()) {
      puesto = iter.next();
      if (puesto.getClass().getSimpleName().equals(cargo)) {
         Salida.mostrar(puesto.toString());
      }
   }
   Salida.separadorGordo();
}
```

```
/**
 * Muestra todos los artesanos y subtipos de Artesano que figuran en el sistema
 */
public static void listarArtesanos() {
   Salida.separadorGordo();
   Salida.mostrar("LISTADO DE ARTESANOS");
   Salida.separadorFino();
   if (!tiene("ArtesanoHora") && !tiene("ArtesanoPlantilla")) {
      Salida.mostrar("No dispone de ningun artesano en su plantilla");
      Salida.separadorGordo();
      return;
   }
   final Iterator<Empleado> iter  = empleados.iterator();
   final Empleado empleado;

   Empleado puesto;
   while (iter.hasNext()) {
      puesto = iter.next();
      if (puesto.getClass().getSimpleName().equals("ArtesanoHora")
      || puesto.getClass().getSimpleName().equals("ArtesanoPlantilla")) {
         Salida.mostrar(puesto.toString());
      }
   }
   Salida.separadorGordo();
}

/**
 * Muestra todos los pedidos de un artesano a seleccionar
 */
public static void verPedidosArtesano() {
   listarArtesanos();

   if (!tiene("ArtesanoHora") && !tiene("ArtesanoPlantilla")) {
      return;
   }

   Salida.mostrar("Introduzca el id del empleado");
   final int id = Entrada.getInt();
   final Artesano empleado = getArtesano(id);
   if (empleado==null) {
      Salida.mostrar("No existe el empleado con id: "+id);
      return;
   }

   RegistroPedido.listarPedidos(Estado.PARADO, empleado);
   RegistroPedido.listarPedidos(Estado.COLA_FABRICACION, empleado);
   RegistroPedido.listarPedidos(Estado.FABRICACION, empleado);
   RegistroPedido.listarPedidos(Estado.FABRICADO, empleado);
```

```
    }

    /**
     * Comprueba si existe algun empleado en el puesto indicado
     * @param cargo El puesto del empleado
     * @return boolean Si tiene empleados en el puesto
     */
    public static boolean tiene(final String cargo) {
        final Iterator<Empleado> iter  = empleados.iterator();
        String puesto;
        while (iter.hasNext()) {
            puesto = iter.next().getClass().getSimpleName();
            if (puesto.equals(cargo)) {
                return true;
            }
        }
        return false;
    }

    /**
     * Muestra un submenú de selección. Permite que un jefe repongan las
     * piezas necesarias para continuar con la fabricación de un pedido
     */
    public static void reponerPiezas() {
        if (!tiene("Jefe")) {
            Salida.mostrar("Necesita tener un jefe en su plantilla");
            return;
        }
        Jefe j;
        int op;
        do {
            listarEmpleados("Jefe");
            Salida.mostrar("Introduzca el ID del jefe o 0 para salir");
            op = Entrada.getInt();
            if (op==0) {
                return;
            }
            j = (Jefe) getEmpleado(op,"Jefe");
        } while(j==null);

        RegistroPedido.listarPedidosPendientesDePiezas();

        if (RegistroPedido.getPedidosPendientesDePiezas().size() == 0) {
            return;
        }

        Salida.mostrar("Seleccione el id del pedido con piezas a reponer");
        final Pedido p =  RegistroPedido.getPedidoPendientePiezas(Entrada.getInt());
```

```
        if (p==null) {
            Salida.mostrar("No existe el pedido con el id introducido");
            return;
        }

        j.reponerPiezas(p);
        Salida.mostrar("Se han repuesto las piezas:");
        RegistroPieza.listarPiezas(p);
    }

    /**
     * Actualiza el estado de un pedido a Estado.PARADO mostrando submenus de selección
     */
    public static void pararPedido() {
        if (!tiene("ArtesanoHora") && !tiene("ArtesanoPlantilla")) {
            Salida.mostrar("No tiene artesanos en su plantilla");
            return;
        }
        if (!RegistroPedido.tienePedidos(Estado.FABRICACION)) {
            Salida.mostrar("No tiene ningun pedido en fabricación por un artesano");
            return;
        }

        Artesano a;
        int op;
        do {
            listarArtesanos();
            Salida.mostrar("Introduzca el ID del artesano o 0 para salir");
            op = Entrada.getInt();
            if (op==0) {
                return;
            }
            a = getArtesano(op);
        } while(a==null);

        Pedido p;
        do {
            RegistroPedido.listarPedidos(Estado.FABRICACION, a);
            Salida.mostrar("Introduzca el ID del artesano o 0 para salir");
            op = Entrada.getInt();
            if (op==0) {
                return;
            }
            p = RegistroPedido.getPedido(op, Estado.FABRICACION, a);
        } while(p==null);

        RegistroPieza.mostrarMenu();
```

```
        final String salir = Entrada.getString();
        if (salir.length() == 1 && salir.charAt(0) == '1'){
            a.detenerTrabajo(p, RegistroPieza.recogerPiezas());
            return;
        }

        a.detenerTrabajo(p);
    }

    /**
     * Selecciona a un comercial y asigna un presupuesto a un pedido gestionado por este.
     * Aparecen submenus para interactuar con la opción seleccionada.
     */
    public static void asignarPresupuesto(){
        if (!tiene("Comercial")) {
            Salida.mostrar("Necesita tener un comercial en su plantilla");
            return;
        }

        if (!RegistroPedido.tienePedidos(Estado.NUEVO)) {
            Salida.mostrar("No tiene ningun pedido pendiente de presupuestos");
            return;
        }

        int op;
        Comercial c;
        do {
            listarEmpleados("Comercial");
            Salida.mostrar("Introduzca el ID del comercial o 0 para salir");
            op = Entrada.getInt();
            if (op==0) {
                return;
            }
            c = (Comercial)getEmpleado(op,"Comercial");
        } while(c==null);
        Pedido p;
        do {
            RegistroPedido.listarPedidos(Estado.NUEVO);
            Salida.mostrar("Introduzca el ID del pedido o 0 para salir");
            op = Entrada.getInt();
            if (op==0) {
                return;
            }
            p = RegistroPedido.getPedido(op, Estado.NUEVO);
        } while(p==null);

        c.establecerPrecio(p);
        Salida.mostrar("Presupuesto asignado correctamente");
```

```
    }

    /**
     * Muestra un submenú para seleccionar un jefe y que este le asigne trabajo a un artesano
     */
    public static void asignarTrabajo(){
        if (!tiene("Jefe")) {
            Salida.mostrar("Necesita tener un jefe en su plantilla");
            return;
        }

        if (!tiene("ArtesanoHora") && !tiene("ArtesanoPlantilla")) {
            Salida.mostrar("No tiene artesanos en su plantilla");
            return;
        }

        if (!RegistroPedido.tienePedidos(Estado.ACEPTADO)) {
            Salida.mostrar("No tiene ningun pedido por asignar a un artesano");
            return;
        }

        Jefe j;

        int op;
        do {
            listarEmpleados("Jefe");
            Salida.mostrar("Introduzca el ID del jefe o 0 para salir");
            op = Entrada.getInt();
            if (op==0) {
                return;
            }
            j = (Jefe) getEmpleado(op,"Jefe");
        } while(j==null);
        Artesano a;
        do {
            listarArtesanos();
            Salida.mostrar("Introduzca el ID del artesano o 0 para salir");
            op = Entrada.getInt();
            if (op==0) {
                return;
            }
            a = getArtesano(op);
        } while(a==null);
        Pedido p;
        do {
            RegistroPedido.listarPedidos(Estado.ACEPTADO);
            Salida.mostrar("Introduzca el ID del pedido o 0 para salir");
            op = Entrada.getInt();
```

```
            if (op==0) {
                return;
            }
            p = RegistroPedido.getPedido(op, Estado.ACEPTADO);
        } while(p==null);

        j.asignarPedido(p,a);
        Salida.mostrar("Pedido asignado correctamente");
    }

    /**
     * Actualiza el pedido de un artesano a seleccionar en un submenú de Estado.FABRICACION a Estado.FABRICADO
     */
    public static void terminarPedidoArtesano(){

        if (!tiene("ArtesanoHora") && !tiene("ArtesanoPlantilla")) {
            Salida.mostrar("No tiene artesanos en su plantilla");
            return;
        }

        int op;
        Artesano a;
        do {
            listarArtesanos();
            Salida.mostrar("Introduzca el ID del artesano o 0 para salir");
            op = Entrada.getInt();
            if (op==0) {
                return;
            }
            a = getArtesano(op);
        } while(a==null);

        Pedido p;
        do {
            RegistroPedido.listarPedidos(Estado.FABRICACION,a);
            Salida.mostrar("Introduzca el ID del pedido o 0 para salir");
            op = Entrada.getInt();
            if (op==0) {
                return;
            }
            p = RegistroPedido.getPedido(op, a);
        } while(p==null);

        a.finalizarTrabajo(p);
            Salida.mostrar("Pedido finalizado por parte del artesano. Se comenzará con la siguiente tarea.");
    }
```

```
    /**
     * Muestra un submenú para actualizar el estado de un pedido a Estado.RECOGER o Estado.ENTREGAR
     */
    public static void notificarClienteEstado() {

        if (!tiene("Comercial")) {
            Salida.mostrar("No tiene ningún comercial en su plantilla");
            return;
        }

        if (RegistroPedido.getPedidos(Estado.FABRICADO).size() == 0) {
            Salida.mostrar("No tiene ningún nuevo pedido cuya fabricación haya sido completada");
            return;
        }

        RegistroPedido.listarPedidos(Estado.FABRICADO);
        int id;
        Pedido p;
        do {
            Salida.mostrar("Introduzca el id del pedido o 0 para salir");
            id = Entrada.getInt();
            if (id == 0) return;
            p = RegistroPedido.getPedido(id, Estado.FABRICADO);
            if (p == null) {
                Salida.mostrar("ID erroneo");
            }
        } while(p==null);

        Comercial c;
        listarEmpleados("Comercial");
        do {
            Salida.mostrar("Introduzca el id del comercial o 0 para salir");
            id = Entrada.getInt();
            if (id == 0) return;
            c = (Comercial) getEmpleado(id, "Comercial");
            if (c == null) {
                Salida.mostrar("ID erroneo");
            }
        } while(c==null);

        do {
            Salida.mostrar("Seleccione una opción:");
            Salida.mostrar("[1] Notificar de entrega");
            Salida.mostrar("[2] Notificar de recogida");
            Salida.mostrar("[0] Cancelar y salir");
            id = Entrada.getInt();
```

```
        } while (id > 2 || id < 0);

        switch (id) {
            case 0:
            return;
            case 1:
            c.notificarClienteEntregar(p);
            break;
            case 2:
            c.notificarClienteRecoger(p);
            break;
        }
        Salida.mostrar("Se ha actualizado el estado del pedido");
    }

    /**
     * Muestra un submenú y lee entradas del usuario para dar de alta a un nuevo empleado en el sistema
     */
    public static void altaEmpleado() {
        int opcion;
        do {
            Salida.mostrarMensajeAltaEmpleado();
            opcion = Entrada.getInt();
        } while (opcion > 4 || opcion < 0);

        switch (opcion) {
            case 0:
            return;
            case 1:
            addJefe();
            break;
            case 2:
            addComercial();
            break;
            case 3:
            addArtesano();
            break;
        }
    }

    /**
     * Da de alta a un nuevo empleado y muestra un mensaje de éxito en la operación
     */
    public static void addEmpleado(final Empleado e) {
        empleados.add(e);
        Salida.mostrar("El empleado: "+e.toString());
        Salida.mostrar("Ha sido añadido al sistema");
```

```
    }

    /**
     * Comprueba si el sistema tiene empleados y pide un id al usuario para eliminar al empleado indicado
     */
    public static void eliminarEmpleado() {
        listarEmpleados();
        if (empleados.size()==0) return;
        Salida.mostrar("Introduzca el id del empleado que quiere eliminar");
        deleteEmpleado(Entrada.getInt());
    }

    /**
     * Comprueba si el sistema tiene empleados y pide un id al usuario para editar al empleado indicado
     */
    public static void editarEmpleado() {
        listarEmpleados();
        if (empleados.size()==0) return;
        Salida.mostrar("Introduzca el id del empleado que quiere editar");
        final int id = Entrada.getInt();
        final Empleado empleado = getEmpleado(id);

        if (empleado==null) {
            Salida.mostrar("No existe el empleado con id: "+id);
            return;
        }

        Salida.mostrar("Inserte el nuevo nombre del empleado");
        Entrada.getString();
        empleado.setNombre(Entrada.getString());
    }

    /**
     * Elimina a un empleado del sistema
     * @param id El ID del empleado
     */
    private static void deleteEmpleado(final int id) {
        final Empleado empleado = getEmpleado(id);
        if (empleado==null) {
            Salida.mostrar("No existe el empleado con id: "+id);
            return;
        }
        empleados.remove(getEmpleado(id));
    }

    /**
```

```
     * Muestra un submenú y lee entradas del usuario para dar de alta a un nuevo
     * artesano en el sistema
     */
    private static void addArtesano() {
        int opcion;
        do {
            Salida.mostrarMensajeAltaArtesano();
            opcion = Entrada.getInt();
        } while (opcion > 3 || opcion < 0);

        switch (opcion) {
            case 0:
            return;
            case 1:
            addArtesanoHora();
            break;
            case 2:
            addArtesanoPlantilla();
            break;
        }
    }

    /**
     * Da de alta a un nuevo jefe en el sistema con el nombre introducido por el
     * usuario
     */
    private static void addJefe() {
        Salida.mostrar("Introduzca el nombre del jefe");
        Entrada.getString();
        final Jefe e = new Jefe(Entrada.getString());
        addEmpleado(e);
    }

    /**
     * Da de alta a un nuevo comercial en el sistema con el nombre introducido por
     * el usuario
     */
    private static void addComercial() {
        Salida.mostrar("Introduzca el nombre del comercial");
        Entrada.getString();
        final Comercial e = new Comercial(Entrada.getString());
        addEmpleado(e);
    }

    /**
     * Da de alta a un nuevo artesano por hora en el sistema con el nombre
     * introducido por el usuario
     */
```

```
    private static void addArtesanoHora() {
        Salida.mostrar("Introduzca el nombre del artesano por hora");
        Entrada.getString();
        final ArtesanoHora e = new ArtesanoHora(Entrada.getString());
        addEmpleado(e);
    }

    /**
     * Da de alta a un nuevo artesano en plantilla en el sistema con el nombre
     * introducido por el usuario
     */
    private static void addArtesanoPlantilla() {
        Salida.mostrar("Introduzca el nombre del artesano en plantilla");
        Entrada.getString();
        final ArtesanoPlantilla e = new ArtesanoPlantilla(Entrada.getString());
        addEmpleado(e);
    }
}
```

35. Clase RegistroMueble

```
package servicio;

import java.util.*;

import modelo.persona.empleado.*;
import modelo.persona.cliente.*;
import modelo.pedido.Pedido;
import modelo.mueble.Mueble;
import modelo.mueble.mesa.*;
import modelo.mueble.silla.*;
import modelo.enumerable.Operacion;
import servicio.RegistroEmpleado;
import servicio.RegistroMueble;
import servicio.RegistroPedido;
import servicio.Entrada;
import servicio.Salida;

/**
 * Clase encargada de almacenar, insertar, editar y eliminar los muebles que figuran en el sistema.
 *
 * @author Arturo Barba
 */
public class RegistroMueble
{
    /**
     * Variable estática con el listado de todos los muebles que figuran en el sistema
```

```
     */
    public static List<Mueble> muebles = new ArrayList<Mueble>();

    /**
     * Getter Mueble. Devuelve un Mueble dado su ID.
     * @param id El id
     */
    public static Mueble getMueble(int id) {
        Iterator<Mueble> iter  = muebles.iterator();
        boolean encontrado = false;
        Mueble mueble;
        while (iter.hasNext() && !encontrado) {
            mueble = iter.next();
            if (id == mueble.getId()) {
                encontrado = !encontrado;
                return mueble;
            }
        }
        return null;
    }

    /**
     * Lista todos los muebles que figuran en el sistema
     */
    public static void listaMuebles() {
        Iterator<Mueble> iter  = muebles.iterator();
        Mueble mueble;
        while (iter.hasNext()) {
            Salida.mostrar(iter.next().toString());
        }
    }

    /**
     * Añade un mueble al sistema
     * @param b El mueble
     */
    public static void addMueble(Mueble b) {
        muebles.add(b);
    }

    /**
      * Muestra un submenú y requiere que el usuario seleccione una opción para añadir muebles al
sistema
     */
    public static void addMuebles() {
        int opcion;
        do {
            Salida.mostrarMensajeMuebles();
```

```
            opcion = Entrada.getInt();
        } while ( opcion > 2 || opcion < 0);

        if (opcion == 0) return;

        switch (opcion) {
            case 1:
            addMesas();
            break;
            case 2:
            addSillas();
            break;
            default:
            return;
        }
    }

    /**
     * Muestra el listado de muebles en stock del sistema y pide la selección de uno
     * para eliminarlo del sistema
     */
    public static void eliminarMueble() {
        listaMuebles();
        if (muebles.size() == 0)
            return;
        Salida.mostrar("Introduzca el id del mueble que quiere eliminar");
        deleteMueble(Entrada.getInt());
    }

    /**
      * Muestra un submenú y requiere que el usuario seleccione una opción para añadir mesas al sistema
     */
    private static void addMesas() {
        int opcion;
        do {
            Salida.mostrarMensajeMesas();
            opcion = Entrada.getInt();
        } while ( opcion > 3 || opcion < 0);

        if (opcion == 0) return;

        switch (opcion) {
            case 1:
            int tipo;
            do {
                Salida.mostrarMensajeMesaCafe();
                tipo = Entrada.getInt();
```

```
        } while ( tipo > 2 || opcion < 0);

        if (tipo == 1){
            MesaCafeCristal mesa = new MesaCafeCristal();
            muebles.add(mesa);
            Salida.mostrar("Se ha añadido al sistema: "+mesa.toString());
        } else if (tipo == 2){
            MesaCafeMadera mesa = new MesaCafeMadera();
            muebles.add(mesa);
            Salida.mostrar("Se ha añadido al sistema: "+mesa.toString());
        }
        break;
        case 2:
           MesaDeDormitorio mesa = new MesaDeDormitorio();
           muebles.add(mesa);
           Salida.mostrar("Se ha añadido al sistema: "+mesa.toString());
        break;
        case 3:
           MesaDeComedor m = new MesaDeComedor();
           muebles.add(m);
           Salida.mostrar("Se ha añadido al sistema: "+m.toString());
        break;
        default:
        return;
     }
  }

  /**
    * Muestra un submenú y requiere que el usuario seleccione una opción para añadir mesas al sistema
   */
  private static void addSillas() {
     int opcion;
     do {
        Salida.mostrarMensajeSillas();
        opcion = Entrada.getInt();
     } while ( opcion > 3 || opcion < 0);

     if (opcion == 0) return;

     switch (opcion) {
         case 1:
        int tipo;
        do {
           Salida.mostrarMensajeSillaOficina();
           tipo = Entrada.getInt();
        } while ( tipo > 2 || opcion < 0);
```

```
            if (tipo == 1){
                SillaConRuedas s = new SillaConRuedas();
                muebles.add(s);
                Salida.mostrar("Se ha añadido al sistema: "+s.toString());
            } else if (tipo == 2){
                SillaSinRuedas s = new SillaSinRuedas();
                muebles.add(s);
                Salida.mostrar("Se ha añadido al sistema: "+s.toString());
            }
            break;
            case 2:
               SillaPlegable s = new SillaPlegable();
               muebles.add(s);
               Salida.mostrar("Se ha añadido al sistema: "+s.toString());
            break;
            case 3:
               SillaCocina m = new SillaCocina();
               muebles.add(m);
               Salida.mostrar("Se ha añadido al sistema: "+m.toString());
            break;
            default:
            return;
        }
    }

    /**
     * Método encargado de devolver un valor comprendido en un rango
     * @param min La cota inferior
     * @param max La cota superior
     * @return int El valor
     */
    private static int leerCantidad(int min, int max) {
        int cantidad;
        do {
            Salida.mostrar("Entre "+min+" y "+max+".");
            cantidad = Entrada.getInt();
        } while ( cantidad > max || cantidad < min);
        return cantidad;
    }

    /**
     * Elimina un mueble del sistema. Muestra un mensaje de éxito o error.
     * @param id El ID del mueble
     */
    private static void deleteMueble(int id) {
        Mueble mueble = getMueble(id);
        if (mueble==null) {
            Salida.mostrar("No existe el mueble con id: "+id);
```

```
            return;
        }
        muebles.remove(getMueble(id));
    }
}
```

36. Clase RegistroPedido

```
package servicio;

import java.util.*;

import modelo.persona.empleado.*;
import modelo.persona.cliente.*;
import modelo.pedido.Pedido;
import modelo.mueble.Mueble;
import modelo.mueble.mesa.*;
import modelo.mueble.silla.*;
import modelo.enumerable.*;
import servicio.RegistroEmpleado;
import servicio.RegistroMueble;
import servicio.RegistroPedido;
import servicio.Entrada;
import servicio.Salida;

/**
 * Clase encargada de almacenar, insertar los pedidos que figuran en el sistema.
 *
 * @author Arturo Barba
 */
public class RegistroPedido
{
    /**
     * Variable estática con el listado de todos los pedidos
     */
    public static List<Pedido> pedidos = new ArrayList<Pedido>();

    /**
     * Devuelve un pedido
     * @param id El ID
     * return Pedido El pedido
     */
    public static Pedido getPedido(int id) {
        Iterator<Pedido> iter  = pedidos.iterator();
        boolean encontrado = false;
        Pedido p;
        while (iter.hasNext() && !encontrado) {
```

```
            p = iter.next();
            if (id == p.getId()) {
                encontrado = !encontrado;
                return p;
            }
        }
        return null;
    }

    /**
     * Devuelve un pedido en un estado determinado
     * @param id El ID del pedido
     * @param estado El estado del pedido
     * @return Pedido El pedido
     */
    public static Pedido getPedido(int id, Estado estado) {
        Iterator<Pedido> iter  = pedidos.iterator();
        boolean encontrado = false;
        Pedido p;
        while (iter.hasNext() && !encontrado) {
            p = iter.next();
            if (id == p.getId() && p.getEstado() == estado) {
                encontrado = !encontrado;
                return p;
            }
        }
        return null;
    }

    /**
     * Devuelve un pedido elaborado por un artesano en cualquier estado
     * @param id El ID del pedido
     * @param artesano El artesano
     * @return Pedido El pedido
     */
    public static Pedido getPedido(int id, Artesano a) {
        Iterator<Pedido> iter  = pedidos.iterator();
        Pedido p;
        while (iter.hasNext()) {
            p = iter.next();
            if (id == p.getId() && p.getArtesano() == a) {
                return p;
            }
        }
        return null;
    }

    /**
```

```
    * Devuelve un pedido de un cliente en un estado
    * determinado
    * @param id El ID del pedido
    * @param estado El estado del pedido
    * @param c El cliente
    * @return Pedido El pedido
    */
   public static Pedido getPedido(int id, Estado estado, Cliente c) {
      Iterator<Pedido> iter  = pedidos.iterator();
      Pedido p;
      while (iter.hasNext()) {
         p = iter.next();
         if (id == p.getId() && p.getEstado() == estado
         && p.getCliente() == c) {
            return p;
         }
      }
      return null;
   }

   /**
    * Devuelve un pedido elaborado por un artesano en un estado
    * determinado
    * @param id El ID del pedido
    * @param estado El estado del pedido
    * @param artesano El artesano
    * @return Pedido El pedido
    */
   public static Pedido getPedido(int id, Estado estado, Artesano a) {
      Iterator<Pedido> iter  = pedidos.iterator();
      Pedido p;
      while (iter.hasNext()) {
         p = iter.next();
         if (id == p.getId() && p.getEstado() == estado
         && p.getArtesano() == a) {
            return p;
         }
      }
      return null;
   }

   /**
    * Devuelve un pedido en estado PARADO y pendiente de peizas
    * @param id El ID del pedido
    * @param estado El estado del pedido
    * @return Pedido El pedido
    */
   public static Pedido getPedidoPendientePiezas(int id) {
```

```
        Iterator<Pedido> iter  = pedidos.iterator();
        boolean encontrado = false;
        Pedido p;
        while (iter.hasNext() && !encontrado) {
            p = iter.next();
            if (id == p.getId() && p.getEstado() == Estado.PARADO &&
                p.getPiezas().size() > 0) {
                encontrado = !encontrado;
                return p;
            }
        }
        return null;
    }

    /**
     * Devuelve el pedido que ha sido añadido antes al sistema
     * @param List<Pedido> Los pedidos a filtrar
     * @return Pedido El pedido mas longevo
     */
    public static Pedido getSiguientePedido(List<Pedido> p) {
        if (p.size()==0) return null;
        int i, pos = 0;
        for (i = 0; p.size() > i; i++) {
            if (p.get(pos).getId() > p.get(i).getId()) {
                pos = i;
            }
        }
        return p.get(pos);
    }

    /**
     * Devuelve todos los pedidos
     * @return List<Pedido> Los pedidos
     */
    public static List<Pedido> getPedidos() {
        return pedidos;
    }

    /**
     * Devuelve todos los pedidos en un estado determinado
     * @param estado El estado del pedido
     * @return List<Pedido> Los pedidos
     */
    public static List<Pedido> getPedidos(Estado e) {
        ArrayList<Pedido> ps = new ArrayList<Pedido>();
        Iterator<Pedido> iter  = pedidos.iterator();
        boolean encontrado = false;
        Pedido p;
```

```
        while (iter.hasNext() && !encontrado) {
            p = iter.next();
            if (e == p.getEstado()) {
                ps.add(p);
            }
        }
        return ps;
    }

    /**
     * Devuelve todos los pedidos de un cliente en un estado
     * determinado
     * @param estado El estado del pedido
     * @param c El Cliente
     * @return List<Pedido> Los pedidos
     */
    public static List<Pedido> getPedidos(Estado estado, Cliente c) {
        ArrayList<Pedido> ps = new ArrayList<Pedido>();
        Iterator<Pedido> iter  = pedidos.iterator();
        Pedido p;
        while (iter.hasNext()) {
            p = iter.next();
            if (p.getCliente() == c && p.getEstado() == estado) {
                ps.add(p);
            }
        }
        return ps;
    }

    /**
     * Devuelve todos los pedidos elaborados por un artesano en un estado
     * determinado
     * @param estado El estado del pedido
     * @param artesano El artesano
     * @return List<Pedido> Los pedidos
     */
    public static List<Pedido> getPedidos(Estado estado, Artesano a) {
        ArrayList<Pedido> ps = new ArrayList<Pedido>();
        Iterator<Pedido> iter = pedidos.iterator();
        Pedido p;
        for (int i = 0; pedidos.size() > i; i++) {
            p = pedidos.get(i);
            if (p.getArtesano() == a && p.getEstado() == estado) {
                ps.add(p);
            }
        }
        return ps;
    }
```

```
/**
 * Devuelve un listado con todos los pedidos parados pendientes de piezas
 * @return List<Pedido> La lista de pedidos
 */
public static List<Pedido> getPedidosPendientesDePiezas(){
    List<Pedido> p = getPedidos(Estado.PARADO);
    List<Pedido> pedidosSinPiezas = new ArrayList<Pedido>();
    for (int i = 0; p.size() > i; i++) {
        if (p.get(i).getPiezas().size() > 0) {
            pedidosSinPiezas.add(p.get(i));
        }
    }
    return pedidosSinPiezas;
}

/**
 * Indica si figuran pedidos en el sistema
 * @return true|false si tiene pedidos
 */
public static boolean tienePedidos() {
    return pedidos.size() > 0;
}

/**
 * Indica si figuran pedidos en el sistema en un estado
 * @param e El estado
 * @return true|false si tiene pedidos
 */
public static boolean tienePedidos(Estado e) {
    return getPedidos(e).size() > 0;
}

/**
 * Muestra todos los pedidos
 */
public static void listarPedidos(){
    Salida.separadorGordo();
    Salida.mostrar("LISTA DE PEDIDOS");
    Salida.separadorFino();
    if (pedidos.size() == 0) {
        Salida.mostrar("No existe ningún pedido");
        Salida.separadorGordo();
        return;
    }

    for (int i = 0; Estado.values().length > i; i++) {
      listarPedidos(Estado.values()[i]);
```

```
        }
    }

    /**
     * Muestra todos los pedidos en un estado determinado
     * @param estado El estado del pedido
     */
    public static void listarPedidos(Estado estado){
        Salida.salto();
        Salida.separadorGordo();
        Salida.mostrar("PEDIDOS EN ESTADO "+estado.toString().toUpperCase());
        Salida.separadorFino();
        if (getPedidos(estado).size() == 0) {
            Salida.mostrar("No existe ningún pedido en el estado "
                +estado.toString().toLowerCase());
            Salida.separadorGordo();
            return;
        }
        List<Pedido> p = getPedidos(estado);
        for (int i = 0; p.size() > i; i++) {
            Salida.mostrar(p.get(i).toString());
        }
        Salida.separadorGordo();
    }

    /**
     * Muestra todos los pedidos realizados por un cliente en un estado
     * determinado
     * @param estado El estado del pedido
     * @param cliente El cliente
     */
    public static void listarPedidos(Estado estado, Cliente c) {
        List<Pedido> p = getPedidos(estado, c);
        Salida.salto();
        Salida.separadorGordo();
        Salida.mostrar("PEDIDOS EN ESTADO "+estado.toString().toUpperCase());
        Salida.separadorFino();
        if (p.size() == 0) {
            Salida.mostrar("No existen pedidos con estas características");
            return;
        }
        for (int i = 0; p.size() > i; i++) {
            Salida.mostrar(p.get(i).toString());
        }
        Salida.separadorGordo();
    }

    /**
```

```
     * Muestra todos los pedidos elaborados por un artesano en un estado
     * determinado
     * @param estado El estado del pedido
     * @param artesano El artesano
     */
    public static void listarPedidos(Estado estado, Artesano a) {
        List<Pedido> p = getPedidos(estado, a);
        Salida.salto();
        Salida.salto();
        Salida.separadorGordo();
        Salida.mostrar("PEDIDOS EN ESTADO "+estado.toString().toUpperCase());
        Salida.separadorFino();
        if (p.size() == 0) {
            Salida.mostrar("No existen pedidos con estas características");
            return;
        }
        for (int i = 0; p.size() > i; i++) {
            Salida.mostrar(p.get(i).toString());
        }
        Salida.separadorGordo();
    }

    /**
     * Muestra todos los pedidos parados pendientes de piezas
     */
    public static void listarPedidosPendientesDePiezas(){
        Salida.salto();
        Salida.separadorGordo();
        Salida.mostrar("PEDIDOS  PARADOS POR FALTA DE PIEZAS");
        Salida.separadorFino();
        List<Pedido> p = getPedidosPendientesDePiezas();
        if (p.size() == 0) {
            Salida.mostrar("No hay pedidos pendientes de piezas");
            Salida.separadorGordo();
            return;
        }

        for (int i = 0; p.size() > i; i++) {
            if (p.get(i).getPiezas().size() > 0) {
                Salida.mostrar(p.get(i).toString());
            }
        }

        Salida.separadorGordo();
    }

    /**
     * Muestra un submenú de opciones con tipos de Estado
```

```
 */
public static void listarEstadosPedido() {
    int index;
    for (int i = 0; i < Estado.values().length; i++) {
        index = i + 1;
        Salida.mostrar("["+index+"] " + Estado.values()[i].toString());
    }
}

/**
 * Muestra un menú de selección de opciones con todos los estados
 * disponibles de un pedido y muestra todos los pedidos con el estado
 * seleccionado
 */
public static void listarPedidosEstado() {
    listarEstadosPedido();
    Salida.mostrar("[0] Salir");
    int op = Entrada.getInt(0, Estado.values().length);
    if (op == 0) return;
    listarPedidos(Estado.values()[op - 1]);
}

/**
 * Muestra un menú de opciones y realiza una búsqueda de pedidos por
 * estados en el sistema en función de la opción seleccionada
 */
public static void menuBuscarPedido() {
    listarEstadosPedido();
    Salida.mostrar("[0] Salir");
    int op = Entrada.getInt(0, Estado.values().length);
    if (op==0) return;
    buscarPedido(Estado.values()[op-1]);
}

/**
 * Muestra la información de un pedido seleccionado
 */
public static void buscarPedido() {
    listarPedidos();
    if (!tienePedidos()) return;
    Salida.mostrar("Introduzca el id del pedido");
    Pedido p = getPedido(Entrada.getInt());
    if (p==null) {
        Salida.mostrar("No existe ningun pedido con ese ID");
        return;
    }
    Salida.mostrar(p.toString());
}
```

```
/**
 * Muestra la información de un pedido seleccionado en un estado
 * @param e El estado
 */
public static void buscarPedido(Estado e) {
   listarPedidos(e);
   if (!tienePedidos(e)) return;
   Salida.mostrar("Introduzca el id del pedido");
   Pedido p = getPedido(Entrada.getInt(), e);
   if (p==null) {
      Salida.mostrar("No existe ningun pedido con ese ID en ese estado");
      return;
   }
   Salida.mostrar(p.toString());
}

/**
 * Permite que un cliente que disponga de pedidos en el estado ENTREGAR o
 * RECOGER confirme la recepción de este y se actualicen los datos del
 * sistema
 */
public static void finalizarPedido() {
   if(!RegistroCliente.tieneClientes()) {
      Salida.mostrar("No existen clientes en el sistema");
      return;
   }

   if(!tienePedidos(Estado.ENTREGAR)||!tienePedidos(Estado.RECOGER)) {
      Salida.mostrar("No existen pedidos por recoger ni por entregar");
      return;
   }

   int id;
   Cliente cliente;
   do {
      Salida.mostrar("Introduzca el id del cliente o 0 para salir");
      id = Entrada.getInt();
      if (id == 0) return;
      cliente = RegistroCliente.getCliente(id);
   } while(cliente==null);

   listarPedidos(Estado.RECOGER, cliente);
   listarPedidos(Estado.ENTREGAR, cliente);
   if (getPedidos(Estado.RECOGER, cliente).size() == 0 &&
   getPedidos(Estado.ENTREGAR, cliente).size() == 0
   ) {
      Salida.mostrar("No tiene pedidos por recoger ni entregar");
```

```
            return;
        }

        Pedido p;
        do {
            Salida.mostrar("Introduzca el id del pedido o 0 para salir");
            id = Entrada.getInt();
            if (id == 0) return;
            p = getPedido(id,Estado.ENTREGAR,cliente);
            if (p == null)
                p = getPedido(id,Estado.RECOGER,cliente);
            if (p == null) {
                Salida.mostrar("ID erroneo");
            }
        } while(p==null);

        Salida.mostrar("Se ha confirmado la recepcion del pedio "+p.toString());
        p.setEstado(Estado.FINALIZADO);
    }

    /**
     * Permite que un cliente pueda realizar un nuevo pedido.
     * Muestra varios submenús y añade un nuevo pedido al sistema
     */
    public static void anotarPedido() {
        ArrayList <Mueble> productos = new ArrayList <Mueble>();

        if (!RegistroCliente.tieneClientes()) {
            Salida.mostrar("No hay clientes en el sistema, registrese para iniciar un pedido");
            return;
        }

        int id;
        Cliente cliente;
        RegistroCliente.listarClientes();
        do {
            Salida.mostrar("Introduzca el id del cliente o 0 para salir");
            id = Entrada.getInt();
            if (id == 0) return;
            cliente = RegistroCliente.getCliente(id);
            if (cliente == null) {
                Salida.mostrar("ID erroneo");
            }
        } while(cliente==null);

        int opcion;
        do {
            do {
```

```
            Salida.mostrarOpcionesPedido();
            opcion = Entrada.getInt();
        } while (
        (opcion == 2 && productos.size() == 0)
        || opcion > 2 || opcion < 0);

        if (opcion == 0) return;

        switch (opcion) {
            case 1:
            productos = addProductos(productos);
            Salida.mostrar("Se ha añadido el mueble a la cesta");
            Salida.salto();
            break;
            case 2:
            addPedido(new Pedido(productos, cliente));
            Salida.mostrar("Se ha realizado el pedido correctamente");
            Salida.salto();
            return;
            default:
            return;
        }

    } while (opcion == 1);
}

/**
 * Añade un pedido al sistema
 * @param p El pedido
 */
public static void addPedido(Pedido p) {
    pedidos.add(p);
}

/**
 * Método auxiliar en anotarPedido().
 * Muestra un submenú y recoge datos para añadir muebles a un pedido
 */
private static ArrayList<Mueble> addProductos(ArrayList<Mueble> p) {
    Salida.mostrarMensajeMuebles();
    int opcion = Entrada.getInt(0, 2);
    switch (opcion) {
        case 1:
        Salida.mostrarMensajeSillas();
        p = addSilla(p);
        break;
        case 2:
        Salida.mostrarMensajeMesas();
```

```
            p = addMesa(p);
            default:
            return p;
        }
        return p;
    }

    /**
     * Método auxiliar en addProductos().
     * Muestra un submenú y recoge datos para añadir sillas a un pedido
     */
    private static ArrayList<Mueble> addSilla(ArrayList<Mueble> p){
        int opcion = Entrada.getInt(0, 3);
        switch (opcion) {
            case 1:
            Salida.mostrarMensajeSillaOficina();
            opcion = Entrada.getInt(0, 3);
            if (opcion == 1) {
                p.add(new SillaConRuedas());
            } else if (opcion == 2) {
                p.add(new SillaSinRuedas());
            } else {
                return p;
            }
            break;
            case 2:
            p.add(new SillaCocina());
            case 3:
            p.add(new SillaPlegable());
            default:
            return p;
        }
        return p;
    }

    /**
     * Método auxiliar en addProductos().
     * Muestra un submenú y recoge datos para añadir mesas a un pedido
     */
    private static ArrayList<Mueble> addMesa(ArrayList<Mueble> p){
        int opcion = Entrada.getInt(0, 3);
        switch (opcion) {
            case 1:
            Salida.mostrarMensajeMesaCafe();
            opcion = Entrada.getInt(0, 3);
            if (opcion == 1) {
                p.add(new MesaCafeCristal());
            } else if (opcion == 2) {
```

```
                p.add(new MesaCafeMadera());
            } else {
                return p;
            }
            break;
            case 2:
            p.add(new MesaDeDormitorio());
            case 3:
            p.add(new MesaDeComedor());
            default:
            return p;
        }
        return p;
    }
}
```

37. Clase RegistroPieza

```
package servicio;

import java.util.*;
import modelo.enumerable.*;
import modelo.pedido.*;

/**
 * Clase encargada la gestión de piezas.
 *
 * @author Arturo Barba
 */
public class RegistroPieza
{
    /**
     * Muestra un listado con todas las posibles piezas
     */
    public static void listarPiezas() {
        for (int i = 0; Pieza.values().length > i; i++) {
            Salida.mostrar("["+i+1+"] "+Pieza.values()[i].toString());
        }
    }

    /**
     * Muestra un listado con todas las posibles piezas
     */
     public static void listarPiezas(Pedido p) {
        for (int i = 0; p.getPiezas().size() > i; i++) {
            Salida.mostrar("["+i+1+"] "+p.getPiezas().get(i).toString());
        }
```

```
    }

    /**
     * Muestra un listado con todas las posibles piezas
     */
    public static void mostrarMenu() {
        Salida.salto();
        Salida.mostrar("[1] Notificar que faltan piezas");
        Salida.mostrar("Pulse cualquier otra letra si no faltan piezas");
    }

    /**
     * Muestra un listado con todas las posibles piezas
     * y devuelve una lista con toda las piezas seleccionadas
     */
    public static ArrayList<Pieza> recogerPiezas() {
        ArrayList<Pieza> piezas = new ArrayList<Pieza>();
        Pieza pieza;
        do {
            listarPiezas();
            Salida.mostrar("[0] Continuar y notificar las piezas seleccionadas");
            pieza = leerPieza();
            if (pieza!=null) piezas.add(pieza);
        } while(pieza!=null || (pieza==null&& piezas.size()>0));
        return piezas;
    }

    /**
     * Muestra un listado con todas las posibles piezas
     * y devuelve una lista con toda las piezas seleccionadas
     */
    public static Pieza leerPieza() {
        int id = Entrada.getInt(0,Pieza.values().length-1);
        if (id == 0) {
            return null;
        }
        return Pieza.values()[id-1];
    }
}
```