TEORÍA SEGUNDO PARCIAL PROGRA III

Sources packages;
- Controller
  - Controller.java
- Model
  - Model.java
- View
  - View.java
- Quinielas
  - Quinielas.java

Carpeta:
SRC:
- controller
- model
- view
- quiniela

Proyecto ----------→ Vista ---------→ Controller ----------→ Model
Main ←--------- ←--------- ←---------

OBJETO A
Get ← leyendo
Set → escribiendo

MVC
Un patrón es un esquema de programación que se repite en múltiples aplicaciones de distintas funcionalidades. Es frecuente encontrar patrones en aplicaciones cuya interface de usuario, grafica, es compleja. En particular, el patrón MVC (o su derivado MVVC) se aplica en muchas de las IGU (interfaces gráficas de usuario) o GUI (graphical user interface) que existen en la actualidad. Ayuda a la reutilización del software. Facilita el desarrollo de software "testable". Favorece la aplicación de principios SOLID.

MODELO: Es la parte del programa que maneja los datos. Suele contener una o más colecciones o tablas, y es responsable de sincronizar los archivos locales con uno o más servidores remotos. El software que se ejecuta en esos servidores remotos son las primitivas que usa el modelo. Al software del servidor remoto se le llama en conjunto Back-end.

```java
package data;

public class Model implements SkelInterface {

    private final int[] numbers;
    private boolean datosLeidos, calculoHecho;

    public boolean isDatosLeidos() {
        return datosLeidos;
    }

    public boolean isCalculoHecho() {
        return calculoHecho;
    }

    public Model() {
       numbers = new int[3];
       datosLeidos = false;
       calculoHecho = false;
    }

    @Override
    public void calcularSuma() {
       numbers[2] = numbers[0] + numbers[1];
       calculoHecho = true;
    }

    @Override
    public void guardarDatos(int n1, int n2) {
       numbers[0] = n1;
        numbers[1] = n2;
       datosLeidos = true;
       calculoHecho = false;
    }

    @Override
    public int valorDeLaSuma() {
        return numbers[2];
    }

} // End of class Model
```

VISTA: Es la parte del programa que se comunica con el usuario, tanto para recibir instrucciones como para mostrar la información solicitada. Suele contener varias perspectivas, y un mecanismo de navegación que permite pasar fácilmente de una a otra, o volver a la perspectiva inicial. El software de la vista, en su conjunto, recibe el nombre de Front-End.

```
package view;

import static com.coti.tools.DiaUtil.*;
import controller.Controller;
import static java.lang.System.out;

public class View {

    Controller controlador = new Controller();

    public void runMenu(String menu) {
        boolean terminado = false;
        String opcionSinAcento;
        do {
            opcionSinAcento = readString(menu).toLowerCase();
            switch (opcionSinAcento) {
                case "1" -> this.leerDatos();
                case "2" -> this.hacerCalculos();
                case "3" -> this.mostrarResultados();
                case "q" -> terminado = siOno("%nDesea salir? ");
                default ->  out.printf("%n%nOpción Incorrecta%n%n");
            } // End of switch
        } while (!terminado); // End of do while
    } // End of runMenu()

    private void leerDatos() {
       int n1, n2;
        clear();
        System.out.printf("%n%nPor favor escriba dos enteros:%n%n");
        n1 = readInt("Escriba el primer número  : ");
        n2 = readInt("Escriba el segundo número : ");
        controlador.guardarEnModelo(n1, n2);
    } // End of leerDatos()

    private void hacerCalculos() {
        if (controlador.datosLeidos()) {
            controlador.calcularLaSuma();
            clear();
            out.printf("%n%nSe ha calculado la suma de los números.%n%n");
        } else {
            System.out.printf("%nError: no se han leído los datos%n");
        }

    } // End of hacerCalculos()

    private void mostrarResultados() {
        if (controlador.calculosHechos()) {
            int resultado = controlador.pedirResultadoAlModelo();
            clear();
            out.printf("%n%nLa suma de los números es %4d%n%n", resultado);
        } else {
            System.out.printf("%nError: calcule los resultados%n");
        }
    } // End of mostrarResultados()

} // End of class View
```

CONTROLADOR: Es la parte del programa que comunica la vista con el modelo. El patrón MVC requiere que la vista y el modelo estén completamente desacoplados, esto es, que se pueda modificar la implementación de la vista y la del modelo de forma completamente independiente. El controlador es el mecanismo de comunicación entre ambos, y normalmente contiene la lógica de negocio (el código que modifica el contenido del modelo por indicación de la vista, y que pasa a la vista los resultados de consultar o modificar el modelo).

```
package controller;

import data.Model;

public class Controller {

    Model modelo = new Model();

    public void guardarEnModelo(int n1, int n2) {
        modelo.guardarDatos(n1, n2);
    } // End of method guardarEnModelo()

    public void calcularLaSuma() {
        modelo.calcularSuma();
    } // End of method calcularLaSuma()

    public int pedirResultadoAlModelo() {
        return modelo.valorDeLaSuma();
    } // End of method pedirResultadoAlModelo()

    public boolean datosLeidos() {
        return modelo.isDatosLeidos();
    }

    public boolean calculosHechos() {
        return modelo.isCalculoHecho();
    }

} // End of class Controller
```

INTERFACE

Esta arquitectura de programa, este patrón, se puede aplicar en infinidad de ocasiones. De hecho, es válido siempre que no haya submenús (siempre que una opción del menú de entrada no nos lleve a otro menú con sus propias opciones). Estos se llaman menús modales. largo del curso. El código de la vista se simplifica mucho por usar readInt(). Este método, y muchos otros, forman parte de una colección de clases que se utilizarán a lo largo de curso.

```
package data;

public interface SkelInterface {

  void calcularSuma();

  void guardarDatos(int n1, int n2);

  int valorDeLaSuma();

}
```

LECTURA DE FICHEROS DE TEXTO: Obtener una ruta al fichero (debe ser compatible con cualquier sistema operativo). Obtener una instancia de File para comprobar si existe. Leer el fichero de texto en una colección de Strings. Procesar cada una de esas Strings (podrían estar delimitadas por algún carácter separando cada uno de los campos). Aquí se deberá crear una estructura de datos para almacenar cada uno de los registros leídos. (arrays, colecciones, etc.)

MÉTODOS FACTORY: Factory method es un patrón de diseño creacional que resuelve el problema de crear objetos sin especificar sus clases concretas. El objetivo es crear un método de clase (*static*) que nos permita crear instancias de objetos de esa misma clase a partir de una serie de argumentos pasados a dicho
método.

JAVA SORTING:
Para la ordenación de Arrays o de Collections en Java existen dos interfaces utilizadas frecuentemente. Comparable (más antigua) y Comparator (Java 8). Algunos puntos importantes de cada una:

java.lang.Comparable	java.util.Comparator
int objeto1.compareTo(objeto2)	int compare(objeto1,objeto2)
Se debe **modificar la clase** de las instancias que se desean ordenar.	Se construye una **clase separada** de las instancias que deseas ordenar.
Solo puede realizar **una secuencia de ordenación** (e.g. por nombre)	Se pueden realizar **varias secuencias de ordenación** (e.g. por nombre y apellido)
Se suele encontrar en la API de Java en clases como Calendar, String, Date, etc.	Pensada para ser implementada para ordenar instancias de clases de terceros (clases propias).

Para crear un `Comparator<T>` se pueden emplear dos tipos de sintaxis.

La primera consiste en crear una clase que implementa los métodos de `Comparator<T>`

```
MyComparator implements Comparator<MyClass> {

}
```

La segunda consiste en emplear los métodos `comparing()` y `thenComparing()` de `Comparator<T>`. Esto resulta más cómodo y sencillo de usar, especialmente si se accede los métodos de acceso de la `T` empleando la sintaxis `Clase::getPropiedad`.

sort() de Arrays y Comparator<T>

Utilizaremos la nueva sintaxis de Comparator<T> de Java 8 para ordenar arrays de objetos T empleando los métodos estáticos comparing y thenComparing que ahorran mucho código. Supongamos un array: Alumno [] listaAlumnos.

Esto es una METHOD REFERENCE!

```
//Primero un comparator por nombre
Comparator<Alumno> comparatorNombre = Comparator.comparing(Alumno::getNombre);

//Un comparator por apellido
Comparator<Alumno> comparatorApellido = Comparator.comparing(Alumno::getApellido);

//Comparar primero por nombre y luego por apellido
Comparator<Alumno> comparatorFull = comparatorNombre.thenComparing(comparatorApellido);

//Usamos el Comparator compuesto Collections.sort()
Arrays.sort(listaAlumnos, comparatorFull); // Operación In Place (tiene efecto sobre la colección)
```

sort() de Collections y Comparator<T> **(LAMBDAS)**

Ahora con Lambdas! La ventaja frente a method references (e.g. Alumno::getNombre) es que es posible acceder a los métodos de objetos anidados. Supongamos List<DatosDeAlumno> listaDatosAlumno;

```
public class DatosDeAlumno {

    private Direccion dir;
    private DatosPersonales datosPersonales;
```

Esto es un DatosDeAlumno

Expresión LAMBDA

```
Comparator<DatosDeAlumno> comparatorNombre = Comparator.comparing(dA -> dA.getDatosPersonales().getNombre());

Comparator<DatosDeAlumno> comparatorApellido = Comparator.comparing(dA -> dA.getDatosPersonales().getApellidos());

Comparator<DatosDeAlumno> compuesto = comparatorNombre.thenComparing(comparatorApellido);

Collection.sort(listaDatosAlumno, compuesto);
```

Aquí accedemos a un DatosPersonales y comparamos con su método getApellidos

max() de Collections y Comparator<T>

Obteniendo el mayor de la colección (la edad es un atributo de DatosPersonales).

```
public class DatosDeAlumno {

    private Direccion dir;
    private DatosPersonales datosPersonales;
```

Esto es un: DatosDeAlumno

LAMBDA!

```
Comparator<DatosDeAlumno> porEdad = Comparator.comparing(dA -> dA.getDatosPersonales().getEdad());

Collections.max(listaAlumnos, porEdad);
```

Investigad los métodos disponibles en Arrays/Collections que emplean Comparator

Aquí accedemos a un DatosPersonales y comparamos con su método getEdad

PrintWriter

Clase orientada a escribir en un stream de salida de texto. Creación:

```
String ruta = System.getProperty("user.home") + File.separator +
"Desktop" +"datos.txt";

PrintWriter writer = new PrintWriter(ruta);
```

Métodos interesantes para escribir como:

```
writer.printf("Hola %s", variableNombre);
writer.println("Hola Hola"); //Incluye un salto de línea
```

Finalmente para cerrar el fichero:

```
writer.close()
```

Scanner

Ahora el parámetro de entrada del constructor será un objeto `File`:

```
String ruta = System.getProperty("user.home") + File.separator +
"Desktop" +"datos.txt";
File fileRef = new File(ruta);
Scanner scannerRef = new Scanner(fileRef);
```

Exactamente igual que sucedía antes:

```
XXXX valorLeido = scannerRef.hasXXX();
XXXX valorLeido = scannerRef.nextXXX();
String cadenaLeída = scannerRef.nextLine();
```

Finalmente para cerrar el fichero:

```
scannerRef.close()
```

Clase Files

Lectura de las líneas de un fichero de forma menos “verbosa” y más elegante. Leyendo una **colección** de Strings List<String>.

```
Path ruta = FileSystems.getDefault().getPath("./", "hola.txt");
List<String> lineas = Files.readAllLines(ruta, StandardCharsets.UTF_8);
```

Es posible recorrer esa colección con un for each en Java o un bucle for:

```
for (String linea : lineas){
   System.out.printf(linea);
}
for (int i = 0; i<lineas.size(); i++){
   System.out.printf(líneas.get(i));
}
```

RUTAS

static Path	**pathToDesktop**()	This method returns the path to the Desktop for any platform
static Path	**pathToDocuments**()	This method returns the path to Documents for any platform
static Path	**pathToFileInDocuments** (String nameOfFile)	This method gives back the Path to a file in Documents
static Path	**pathToFileInFolderInDocuments** (String nameOfFolder, String nameOfFile)	This method gives back the Path to a file in a folder in Documents
static Path	**pathToFileInFolderOnDesktop** (String nameOfFolder, String nameOfFile)	This method gives back the Path to a file in a folder on the Desktop
static Path	**pathToFileOnDesktop** (String nameOfFile)	This method gives back the Path to a file on the Desktop
static Path	**pathToFolderInDocuments** (String nameOfFolder)	This method returns a path to a folder in Documents for any platform
static Path	**pathToFolderOnDesktop** (String nameOfFolder)	This method returns a path to a folder on Desktop for any platform

OPMAT

static void	**exportToDisk** (double[][] matrix, File f, String delimiter)	This method exports a double[][] to disk with delimited format
static void	**exportToDisk** (float[][] matrix, File f, String delimiter)	This method exports a float[][] to disk with delimited format
static void	**exportToDisk** (int[][] matrix, File f, String delimiter)	This method exports an int[][] to disk with delimited format
static void	**exportToDisk** (String[][] matrix, File f, String delimiter)	This method exports a String[][] with delimited format and can throw an exception
static String[][]	**importFromDisk** (File f, String delimiter)	This method imports from disk a String[][] written with delimited format
static double[][]	**importFromDisk** (File f, String delimiter, double dummy)	This method imports a double[][] from a delimited file
static float[][]	**importFromDisk** (File f, String delimiter, float dummy)	This method imports a float[][] from a delimited file
static int[][]	**importFromDisk** (File f, String delimiter, int dummy)	This method imports an int[][] from a delimited file

BIBLIOTECAS UTILES

MAIN:

```
import view.View;

View v = new View();
```

VISTA:

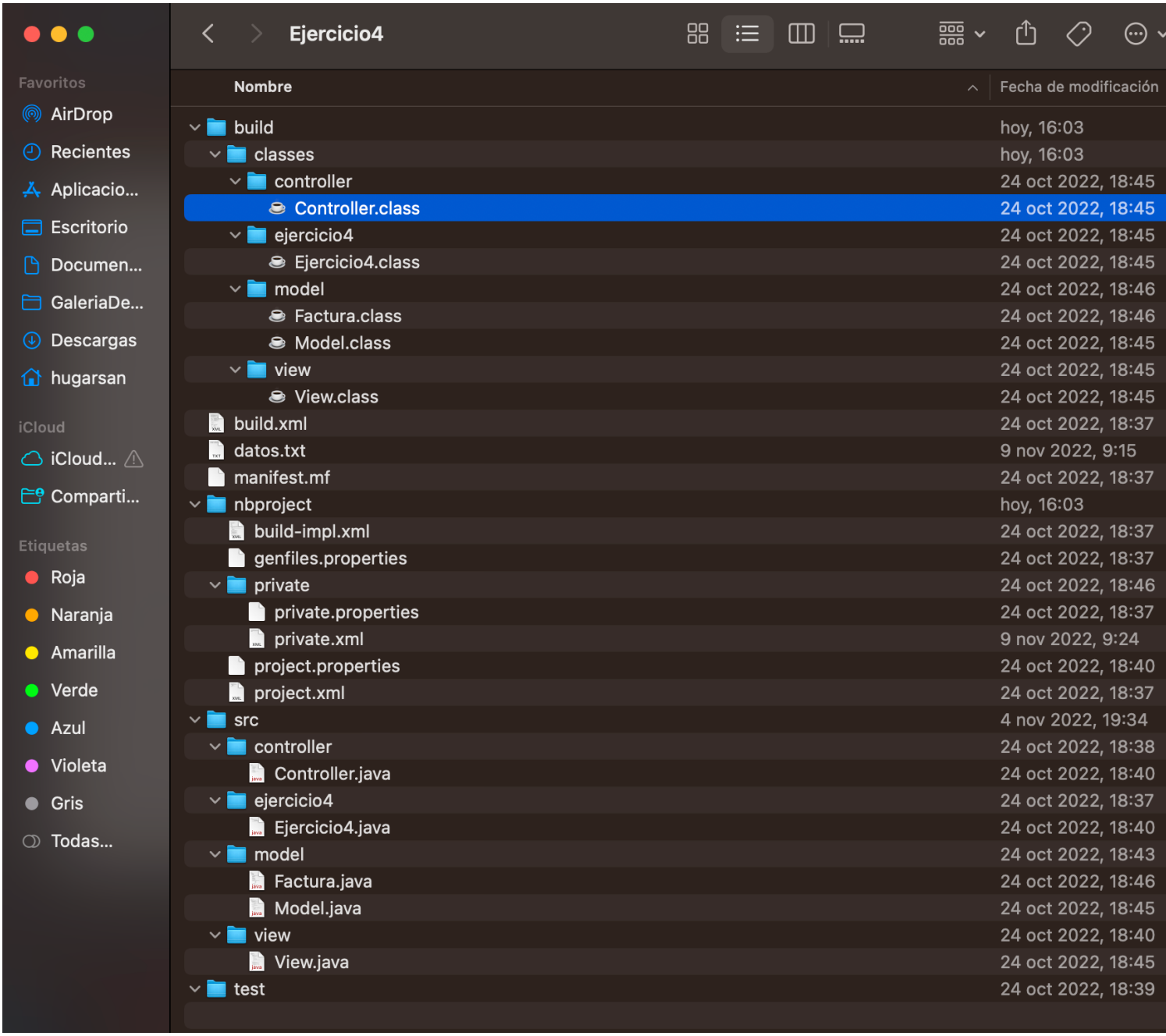
```
package view;

Controller c = new Controller();

import controller.Controller;
import static com.coti.tools.Esdia.*; (readFloat,readString...,readString_ne,yesOrNO...) import static
com.coti.tools.OpMat.printToScreen3;
import java.util.logging.Level; (Para el try, catch de printToScreen3);
import java.util.logging.Logger;

import static java.lang.System.err; (si usamos err.println())
```

CONTROLLER:

```
package controller;

Model m = new Model();

import controller.Controller;
```

MODEL:

```
package model;

import java.util.*; (Comparator, Arrays.sort())
```


Ejercicio4
Favoritos
AirDrop
Recientes
Aplicacio...
Escritorio
Documen...
GaleriaDe...
Descargas
hugarsan
iCloud
iCloud...
Comparti...
Etiquetas
Roja
Naranja
Amarilla
Verde
Azul
Violeta
Gris
Todas...
Nombre
Fecha de modificación
build
hoy, 16:03
classes
hoy, 16:03
controller
24 oct 2022, 18:45
Controller.class
24 oct 2022, 18:45
ejercicio4
24 oct 2022, 18:45
Ejercicio4.class
24 oct 2022, 18:45
model
24 oct 2022, 18:46
Factura.class
24 oct 2022, 18:46
Model.class
24 oct 2022, 18:45
view
24 oct 2022, 18:45
View.class
24 oct 2022, 18:45
build.xml
24 oct 2022, 18:37
datos.txt
9 nov 2022, 9:15
manifest.mf
24 oct 2022, 18:37
nbproject
hoy, 16:03
build-impl.xml
24 oct 2022, 18:37
genfiles.properties
24 oct 2022, 18:37
private
24 oct 2022, 18:46
private.properties
24 oct 2022, 18:37
private.xml
9 nov 2022, 9:24
project.properties
24 oct 2022, 18:40
project.xml
24 oct 2022, 18:37
src
4 nov 2022, 19:34
controller
24 oct 2022, 18:38
Controller.java
24 oct 2022, 18:40
ejercicio4
24 oct 2022, 18:37
Ejercicio4.java
24 oct 2022, 18:40
model
24 oct 2022, 18:43
Factura.java
24 oct 2022, 18:46
Model.java
24 oct 2022, 18:45
view
24 oct 2022, 18:40
View.java
24 oct 2022, 18:45
test
24 oct 2022, 18:39