Unidad 14. Interfaces de Usuario. GUI

Contenido

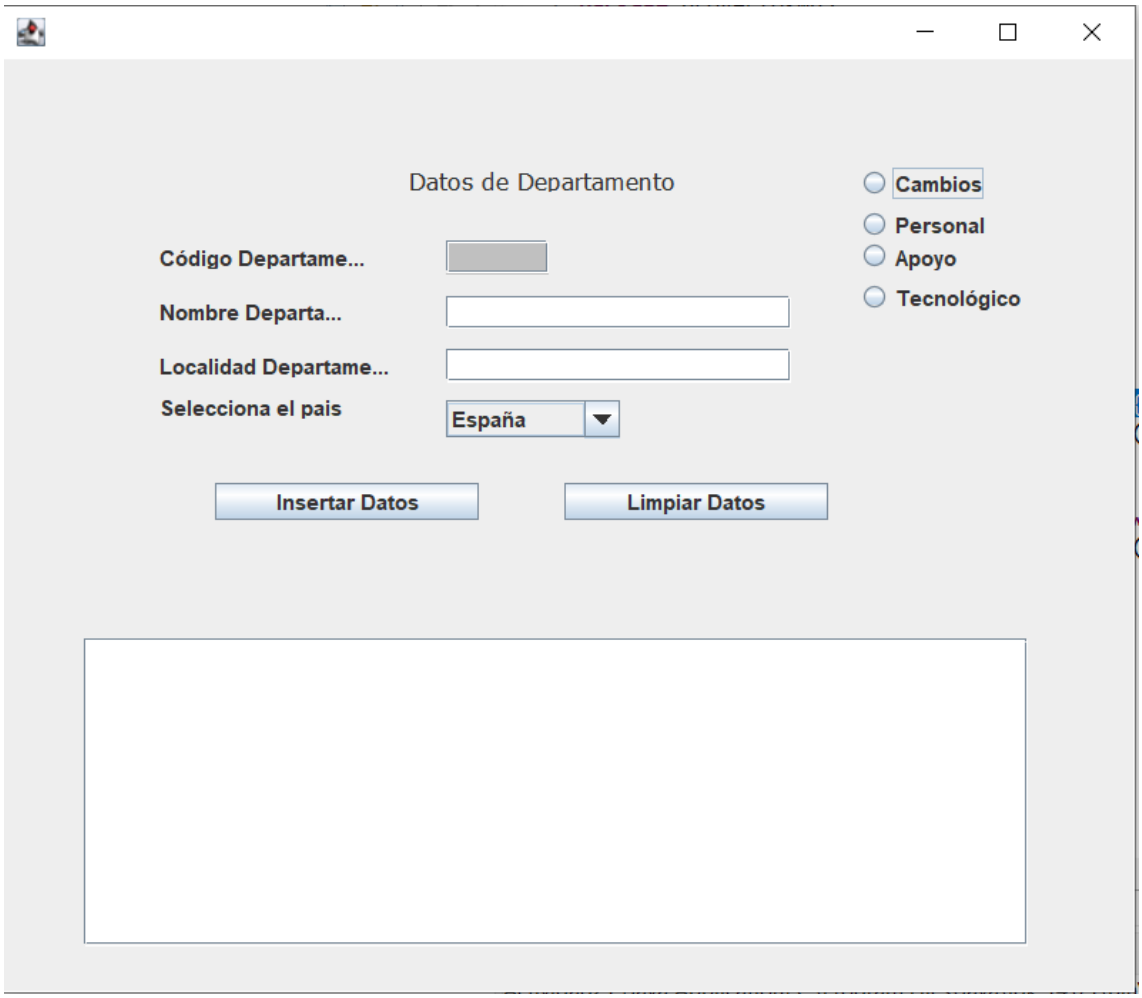
1 Actividad inicial. Autoaprendizaje

Problema del día:

Se les enseña a los alumnos una aplicación de ventanas en Java. Se les da 20 minutos para generar en su IDE un proyecto Java que permita realizar ventanas. Pueden:

1. Investigar en el IDE
2. Buscar por internet

Para realizar la siguiente ventana



2 Introducción

En este tema **vamos a trabajar con estructuras de interfaces gráficas.**. En particular vamos a trabajar con Java Swing que es más fácil de adaptar a los proyectos para Java 14. En el **caso de JavaFX necesitamos** crear **un proyecto Maven y exportar una serie de librerías.**

Además, **en este tema vamos a aplicar lo que hemos aprendido** hasta ahora de manejo con **la API Stream de colecciones de datos y la clase Optional** será importante para **minimizar el manejo de errores y el uso de Excepciones** para ir **mostrando los datos en nuestra interfaz gráfica**. Es **una aplicación practica de los conocimientos adquiridos a las clases de Java Swing**. Se incorporará un **patrón de diseño** nuevo, el **Memento**, y un **modelo de diseño nuevo el MVC**, y mediante un ejemplo iremos viendo **como integrar Java 8-14**, y **patrones de diseño a nuestra interfaz.**

3 Tratamiento de excepciones y uso de Optional

Para **finalizar el tema vamos mediante un ejemplo a mostrar como programar el comportamiento** de **nuestra aplicación** cuando **hay errores usando Optional**. Para ello vamos a **ofrecer un sencillo programa**, en el que **forzaremos las excepciones en el tratamiento de datos** y **mostraremos como debe comportarse nuestra aplicación**. **Usaremos el modelo de Usuario que hemos venido** usando **en los ejemplos** a lo largo del curso. Y creamos una función **Main.java nueva**.

En el **siguiente ejemplo Main.java tenemos tres funciones para dat**os. Las tres **están diseñadas para que fallen a propósito** para **explicar que hacer en caso de un error** en la aplicación. Cada que **intentamos recuperar datos devolvemos los datos** dentro de un **Optional** como podéis ver a continuación.

La **primera carga los datos del sistema**, una **lista de usuarios**. Como veis devuelve un **Optional de tipo lista de usuarios**. Lo **inicializamos como vacío** e **intentamos leer de nuestro fichero de datos**, que **no existe para que falle a propósito**.

```
        Optional<List<Usuario>> optLista = Optional.empty();
        Path file = Path.of("d:\\datos.bin");
```

Si **la lectura del fichero es correcto** devolvemos un **Optional lleno** con la lista.

```
        try {
            byte[] bytes = Files.readAllBytes(file);
```

```
            optLista =Optional.of(listaUsuarios);
            return optLista;
```

Si la lectura no es correcta saltará la excepción IOException. En este caso **escribiremos en el log que hay un error**, pintaremos **el printStackTrace**, y finalmente **devolvemos el Optional vacio**. Es **mucho mejor que devolver null**, que **tiende a propagar más errores**.

```
catch (IOException ex) {
            Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex
);
            ex.printStackTrace();
            return optLista;

        }
```

```
public static Optional<List<Usuario>>cargarDatos() {

        Optional<List<Usuario>> optLista = Optional.empty();
        Path file = Path.of("d:\\datos.bin");



        try {
            byte[] bytes = Files.readAllBytes(file);

           optLista =Optional.of(listaUsuarios);
           return optLista;
        } catch (IOException ex) {
            Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex
);
            ex.printStackTrace();
            return optLista;

        }
```

```
    }
```

Y **ahora vamos a ver nuestro programa principal**. Siempre en **el inicio de un programa principal intentamos cargar datos**. Si **no podemos cargar los datos completos** de **nuestra aplicación** porque **hay un error** lo mejor **es cerrarla**. Esto pasará **si el Optional viene vacío** que será **porque no hemos podido recuperar los datos**. En una **aplicación con interfaz gráfica deberíamos mandar un dialogo de error**.

```
        Optional <List<Usuario>> optLista = Main.cargarDatos();

        if (optLista.isEmpty()) {
            //EN VENTANAS HAY QUE MANDAR UN DIALOGO DE ERROR
            System.out.println("Error en los datos. El programa se cerrará");

            System.exit(0);
        }
```

Para recuperaUsuario realizamos **la misma labor, pero en el programa principal si falla** recuperar un **solo usuario**, **no cerramos la aplicación completa**, escribimos en el **log**, pintamos el **stackTrace** de la **Excepción**, **mandamos un diálogo de error** y le dejamos al **sistema seguir funcionando.** Estudiad el código. Como veis **volvemos a devolver un Optional<Usuario>**. Si el **Optional vuelve vacío es que ha habido un error**. No explicamos aquí el código completo porque es prácticamente igual que el primer caso. Estudiad el código.

```
    public static Optional<Usuario> recuperaUsuario(int index) {
```

Para **guardarUsuario**, igual que **para guardar cualquier dato podemos devolver o un booleano con true si ha ido bien**, y **false si ha ido mal,** o un **valor entero positivo si ha ido bien**, **negativo** si ha ido **mal**, a elección del programador. Si **falla guardar un solo usuario, no cerramos la aplicación completa**, escribimos en el **log**, pintamos el **stackTrace de la Excepción**, mandamos un **diálogo de error** y le dejamos al **sistema seguir funcionando**. No explicamos aquí el código completo porque es prácticamente igual que el primer caso. Estudiad el código.

```
    public static boolean guardaUsuario(Usuario user) {
```

Main.java

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.List;
import java.util.Optional;
import java.util.logging.Level;
import java.util.logging.Logger;
import tarea10.gestionerroresoptonal.modelo.GeneraUsuarios;
import tarea10.gestionerroresoptonal.modelo.Usuario;




/**
 *
 * @author carlo
 */
public class Main {

    public static List<Usuario> listaUsuarios = GeneraUsuarios.devueveUsuario
sLista(10);


    public static Optional<List<Usuario>>cargarDatos() {

        Optional<List<Usuario>> optLista = Optional.empty();
        Path file = Path.of("d:\\datos.bin");



        try {
```

```
            byte[] bytes = Files.readAllBytes(file);

           optLista =Optional.of(listaUsuarios);
           return optLista;
        } catch (IOException ex) {
            Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex
);
            ex.printStackTrace();
            return optLista;

        }

        }

    public static Optional<Usuario> recuperaUsuario(int index) {

    Optional<Usuario> optUsuario = Optional.empty();

            try {

               optUsuario = Optional.of(listaUsuarios.get(index));
                return optUsuario;
            } catch(Exception e) {


                e.printStackTrace();
                return optUsuario;
            }



    }


    public static boolean guardaUsuario(Usuario user) {

           List<Usuario> lista=null;
```

```
        try {

            lista.add(user);
            return true;
          } catch(Exception e) {


              e.printStackTrace();
              return false;
          }


}

 /**
 * @param args the command line arguments
 */
public static void main(String[] args) {
    // TODO code application logic here


    Optional <List<Usuario>> optLista = Main.cargarDatos();

    if (optLista.isEmpty()) {
        //EN VENTANAS HAY QUE MANDAR UN DIALOGO DE ERROR
        System.out.println("Error en los datos. El programa se cerrará");

        System.exit(0);
    }

    Optional<Usuario> user = Main.recuperaUsuario(12);

    if (user.isEmpty()) {
          //EN VENTANAS HAY QUE MANDAR UN DIALOGO DE ERROR
         System.out.println("Error al recuperar un usuario.");
```

```
        }

        if (Main.guardaUsuario(listaUsuarios.get(12))) {

             System.out.println("El usuario se guardo correctamente");


        } else {

             System.out.println("Error al guardar el usuario");

        }

    }

}
```

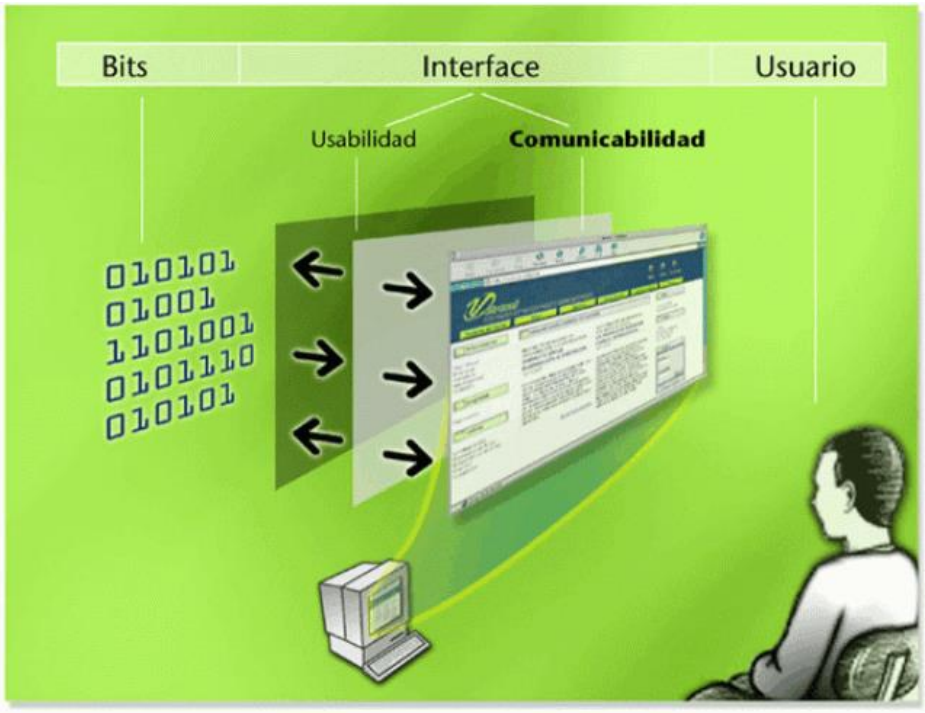
4 Interfaz Gráfica de Usuario (GUI)

La **interfaz gráfica de usuario, conocida también como GUI** (del inglés **graphical user interface**), es un **programa informático que actúa de interfaz de usuario**, utilizando un **conjunto de imágenes y objetos gráficos para representar la información y acciones disponibles en la interfaz**. Su **principal uso consiste en proporcionar un entorno visual sencillo para permitir la comunicación** con el **sistema operativo de una máquina o computador.**

Habitualmente **las acciones se realizan mediante manipulación directa**, para facilitar **la interacción del usuario con la computadora**. Surge como **evolución de las interfaces de línea de comandos** que se usaban para operar **los primeros sistemas operativos y es pieza fundamental en un entorno gráfico**. Como **ejemplos** de interfaz gráfica de usuario, cabe citar los entornos de escritorio **Windows, el X-Window de GNU/Linux o el de Mac OS X, Aqua.**

En el **contexto del proceso de interacción persona-computadora**, la **interfaz**

gráfica de usuario es el artefacto tecnológico de un sistema interactivo que posibilita, a través del **uso y la representación del lenguaje visual**, una **interacción amigable** con un sistema informático.



4.1 Precursores y evolución de las GUI

Los **investigadores del Stanford Research Institute liderados por Douglas Engelbart**, **desarrollaron una interfaz de hipervínculos** en **modo texto** gobernada por **un ratón**, que también **inventaron**. Este **concepto fue ampliado y trasladado al entorno gráfico por los investigadores del Xerox PARC** en la ciudad estadounidense de **Palo Alto**. El entorno se denominó **PARC User Interface** y en él se definieron los **conceptos de ventanas, casilla de verificación, botones de radio, menús y puntero del ratón**. La interfaz fue **implementada comercialmente en el computador Xerox Star 8010**.

Xerox Alto

El **Xerox Alto, desarrollado en el Xerox PARC en 1973**, fue una de las primeras **computadoras personales**, así como el primero que utilizó la metáfora de escritorio y una interfaz gráfica de usuario.

Xerox Star 8010

La estación de trabajo Xerox Star, conocida oficialmente como el 8010 Star Information System (Sistema de Información Estrella 8010) fue introducida por **Xerox Corporation en 1981**. Fue el primer **sistema comercial en incorporar varias tecnologías que han llegado a ser hoy en día corrientes en**

computadores personales, incluyendo la pantalla con bitmaps en lugar de solo texto, una interfaz gráfica de usuario basada en **ventanas, iconos, carpetas, ratón, red Ethernet, servidores de archivos, servidores de impresoras y correo electrónico**.

Apple Lisa, Macintosh, Apple II GS.

Tras una **visita al Xerox PARC en 1979, el equipo de Apple encabezado por Jef Raskin** se concentra en diseñar un **entorno gráfico para su nueva generación de 16 bits,** que se verá **plasmado en el Apple Lisa en 1983**. Ese sistema gráfico es portado al **sucesor del Apple II, el Apple II GS**. Un segundo **equipo trabaja en el Apple Macintosh que verá la luz en 1984** con una **versión mejorada del entorno gráfico del Lisa** ("pretendimos hacer una computadora tan simple de manejar como una tostadora"). Desde ese momento **el Mac reinará como paradigma de usabilidad de un entorno gráfico**; pese a que **por debajo el sistema operativo sufra cambios radicales, los usuarios no avanzados no son conscientes de ello** y no sufren los problemas de otras plataformas.

Workbench

Workbench es el nombre dado por Commodore a la interfaz gráfica del AmigaOS, el sistema operativo del **Commodore Amiga lanzado en 1985**. A diferencia de los sistemas más populares (GEM, Mac OS, MS Windows...) **es un verdadero entorno multitarea solo rivalizado por la interfaz X Window System** de los diferentes **versiones de Unix**. La frase **más repetida por un amiguero es: «para masacrar marcianos, formatear un disquete y enviar o recibir un Fax** todo **a la vez y sin colgarse**, necesitas un 386 con disco duro, 16 MB de RAM y OS/2; un Amiga 500 con disquete y solo su memoria base (512 KB de RAM y 512 KB de ROM) es capaz de todo eso». Aunque **muy popular por los espectaculares (para entonces) gráficos de la máquina y su gran plantel de videojuegos**, será la **negligencia de sus sucesivos propietarios la principal causa de que acabe restringido** a solamente la plataforma Amiga.

Apple y Microsoft

En **1982, Apple había comenzado como una microempresa formada por dos empleados, Steve Jobs y Steve Wozniak**, y había **crecido hasta convertirse en una empresa de 300 millones de dólare**s. En **1983, ya se había convertido en una empresa de 1000 millones de dólares**, el mismo **valor que IBM.**

En **1985, Microsoft saca al mercado Windows 1.0, entorno gráfico para computadoras PC IBM compatible**s, con **muchos parecidos al Mac OS**. La respuesta de **Apple a la introducción del sistema operativo Windows fue la interposición de una demanda de varios miles de millones de dólares** contra Microsoft, por violación de **copyright**.

En **1987 IBM se vio obligada a entrar en el mercado de las computadoras personales con entorno gráfico con su modelo IBM Personal System/2 (PS/2**),

aliándose con Bill Gates (Microsoft), que **había desarrollado el OS/2**. La **interfaz gráfica de este sistema operativo era muy similar a la de Apple**. El **OS/2 no se convirtió en el nuevo estándar** del sector, debido fundamentalmente al **conflicto de intereses entre IBM y Microsoft.**

La aparición de **computadoras IBM clónicas hizo que el sistema Windows se popularizara, lo que restó mercado a Apple**. Esta se recuperó a finales de 1990 lanzando nuevos productos.

Una **señal inequívoca del éxito de Apple fue la aparición de productos similares**: una pequeña compañía **llamada Nutek Computers Inc., anunció que estaba desarrollando una computadora compatible con el Macintosh**.
En 1991, **John Sculley, director de Apple, reveló que la compañía estaba considerando competir contra Microsoft en el campo del software** vendiendo su **sistema operativo a terceros. Apple reveló que estaba manteniendo conversaciones con su antiguo rival, IBM,** destinadas a **compartir tecnologías**. Decidieron crear una **joint venture para desarrollar un sistema operativo avanzado que ambas utilizarían en sus nuevas máquinas** y licenciarían a terceros. Este plan presentaba un **desafío directo a Microsoft.**
Microsoft **consigue convertir a Windows en el sistema operativo más utilizado en el mundo**, dejando a **Apple en un segundo lugar.**
En *la actualidad estas dos compañías APPLE y Microsoft* siguen desarrollando, **sistemas operativos con interfaz gráfica** para **ordenadores personales**.

4.1.1 IPads, Tablets y SmartPhones Apple y Google.

Apple

iPhone es una línea de teléfonos inteligentes de alta gama diseñada y comercializada por **Apple Inc.** Ejecuta el sistema **operativo móvil iOS**, conocido hasta **mediados de 2010 como "iPhone OS".**

i**Pad es una línea de tabletas diseñadas y comercializadas por Apple Inc**. La primera **generación fue anunciada el 27 de enero de 2010, mientras que el 2 de marzo de 2011 apareció la segunda generación**. Se sitúa en una categoría entre un teléfono inteligente y una computadora portátil, enfocado más al acceso que a la creación de aplicaciones y temas.

El **iPhone dispone de cámara de fotos y un reproductor de música** (equivalente al del iPod), además **de software para enviar y recibir mensajes de texto y de voz**. También **ofrece servicios de Internet**, como leer **correo electrónico, cargar páginas web y conectividad por Wi-Fi**. La **primera generación de teléfonos eran GSM cuatribanda con la tecnología EDGE**; la **segunda generación incluía UMTS con HSDPA.;2 la sexta generación ya**

incluía 5G LTE.

iOS es un sistema operativo móvil de la multinacional Apple Inc. Originalmente **desarrollado para el iPhone (iPhone OS)**, después se **ha usado en dispositivos** como **el iPod touch y el iPad**. Apple no permite la instalación de iOS en hardware de terceros.

Google y Android OS

Android es un sistema operativo móvil basado en núcleo Linux y otros software de código abierto. Fue diseñado para **dispositivos móviles con pantalla táctil, como teléfonos inteligentes, tabletas, relojes inteligentes (Wear OS), automóviles** con otros sistemas a través de **Android Auto**, al igual los automóviles con el sistema Android Automotive y **televisores Leanback**.
Inicialmente fue **desarrollado por Android Inc., que adquirió Google en 2005**. Android fue **presentado en 2007 junto con la fundación del Open Handset Alliance** (un consorcio de compañías de hardware, software y telecomunicaciones) para avanzar en los **estándares abiertos de los dispositivos móviles**. El código **fuente principal de Android se conoce como Android Open Source Project** (AOSP), que se licencia principalmente bajo la **Licencia Apache.**5 Android es el sistema **operativo móvil más utilizado del mundo,** con una cuota de mercado superior al 90.

Interfaz gráfica de usuario para smartphones tables

La característica común de ambos sistemas operativos y tecnologías es la del uso de un nuevo elemento de interacción que sustituye al ratón, que es la **pantalla táctil**. Surgen **los diseños PostWin post-WIMP** ("windows, iconos, menus, punteroes") comprende el **trabajo en interfaces de usuario, principalmente interfaces gráficas de usuario**, que intentan ir más allá del **paradigma de ventanas, iconos, menús y un dispositivo señalador**, es decir, **interfaces WIMP**.

Además, el **tamaño de pantallas es más pequeño** con lo que los **diseños de interfaces cambian para adaptarse** a estas **nuevas pantallas**. No existe como con **ordenadores personales o portátiles** un tamaño de p**antalla standard, los tamaños de pantalla son variables y dependientes del fabricante**. Surge el c**oncepto de DPS o densidad de pantalla por pulgada** junto a los diseños de interfaces que se adaptan a este tipo de pantalla, **diseños Responsive.**

G**eneral Magic es en teoria el padre de toda la interfaz gráfica de usuario moderna** para **smart phones**, es decir, **basada en pantalla táctil**, incluyendo el **iPhone**. En **2007, con el iPhone** y más tarde **en 2010 con la introducción del iPad**, Apple **popularizó el estilo de interacción post-WIMP** para pantallas multitáctil, con aquellos dispositivos considerados como hitos en el desarrollo de dispositivos móviles.

Otros **dispositivos portátiles como reproductores de MP3 y teléfonos móviles** han sido un área de **despliegue floreciente para los GUIs en los últimos años**.

Desde mediados de la **década de 2000,** una gran mayoría de los dispositivos **portátiles han avanzado a tener resoluciones y tamaños de pantalla alta**. (La pantalla de 2.560 × de 1.440 píxeles del **Galaxy Note 4** es un ejemplo). Debido a esto, estos **dispositivos tienen sus propias interfaces de usuario famosas y sistemas operativo**s que tienen **grandes comunidades "Caseras"** dedicadas a **crear sus propios elementos visuales**, tales como iconos, menús, fondos de pantalla, y más. Las interfaces p**ost-WIMP se utilizan a menudo en estos dispositivos móviles**, donde los **dispositivos señaladores tradicionales requeridos** por la **metáfora de escritorio** no son prácticos.

A medida **que el hardware gráfico de alta potencia extrae una potencia considerabl**e y **genera un calor significativo**, muchos de **los efectos 3D desarrollados entre 2000 y 2010 no son prácticos** en esta clase de d**isposi**tivo. Esto ha llevado **al desarrollo de interfaces más simples** que hacen **una característica de diseño de bidimensionalidad como la exhibida p**or la interfaz de **usuario de Metro (Moderna)** utilizada por primera vez **en Windows 8** y el **rediseño de Gmail de 2012**.

Incluso **los diseños web han cambiado debido a los móviles y tablets** ya que el **70% de los usuarios web** usando un **móvil o tablet** para navegar. Recomendable **leer y estudiar Materia lDesign**, una compañía **de Google que se dedica a ofrecer componentes de terceros** para **diseño web Web Responsive y de aplicaciones móviles** .

https://material.io/design

Otra compañía que **ofrece librerías para desarrollo de interfaces web es BootStrap**. Diseños y **componentes Web Responsive**, que adaptan la web a **cualquier tipo de pantalla.**
https://getbootstrap.com/

4.1.2 Interfaces ·3D

En **la primera década del siglo XXI**, el rápido **desarrollo de las GPU** (tarjetas gráficas) **condujo a una tendencia para la inclusión de efectos 3D** en el **manejo de ventanas**. Se basa en la **investigación experimental en diseño de interfaces** de usuario tratando de ampliar el **poder expresivo de los kits de herramientas de desarrollo existentes** con el fin de **mejorar las señales físicas** que permiten la **manipulación directa**.
Los nuevos **efectos comunes a varios proyectos son el cambio de tamaño** y **el zoom a escala, varias transformaciones y animaciones de windows** (ventanas tambaleantes, minimización suave a la bandeja del sistema...), **composición de imágenes** (utilizadas para sombras paralelas de ventanas y transparencia) y la **mejora de la organización global de ventanas abiertas** (zoom a escritorios virtuales, cubo de escritorio, Exposé, etc.) El **escritorio BumpTop** de prueba de concepto **combina una representación física de documentos con herramientas para la clasificación de documentos** posibles solo en el **entorno simulado**, como la **reordenación instantánea y la agrupación automatizada**

de **documentos** relacionados.

Estos efectos **se han hecho populares gracias al uso generalizado de tarjetas de vídeo 3D** (principalmente **debido a juegos**) que permiten un **procesamiento visual complejo con bajo uso de CPU**, utilizando **la aceleración 3D** en la **mayoría de las tarjetas gráficas** modernas para dar **el manejo a los usuarios** de la aplicación en **una escena 3D.** La **ventana de la aplicación se extrae** de la **pantalla en un búfer de píxeles y la tarjeta gráfica** la **representa en la escena 3D.**

4.2 Tipos de interfaz gráfica. Clasificación

CUI

Character User Interface (CUI) es un tipo de **interfaz de usuario que utiliza solamente caracteres alfanuméricos y pseudografías** para la **entrada-salida y la presentación de la información**. Se caracteriza por **una baja demanda de recursos de hardware de E/S** (en particular, memoria) y **alta velocidad de visualización** de información. **Interfaces de consola** de comandos son un ejemplo de este tipo de interfaces.

GUI

Una GUI (interfaz gráfica de usuario) es un sistema de componentes visuales interactivos para software informático. Una **GUI muestra objetos que transmiten información y representan las acciones** que puede **realizar el usuario**. Los **objetos cambian de color, tamaño o visibilidad** cuando el **usuario interactúa** con ellos.
Los **objetos GUI incluyen iconos, cursores y botones**. Estos elementos gráficos a veces se mejoran con sonidos, o efectos visuales como transparencia y sombras paralelas.

Las GUI y ZUI

Los **tipos de interfaces gráficas de usuarios** (GUI) que se encuentran **en juegos de computadora, y las GUI avanzados basados en realidad virtual**, se usan con **frecuencia en tareas de investigación**. Muchos grupos de **investigación en Norteamérica y Europa incorporando a productos actualmente en la interfaz de enfoque** del usuario o ZUI (**Zooming User Interface**), que es un adelanto **lógico de las GUI, mezclando 3D con 2D**. Podría expresarse como «**2 dimensiones y media en objetos vectoriales** de una dimensión».

Interfaz de usuario de pantalla táctil

Algunas **GUI son diseñadas para cumplir con los rigurosos requisitos** de los mercados verticales. Estos se **conocen como las GUI de uso específico.** Un ejemplo de un **GUI de uso específico es la ahora familiar pantalla táctil o**

touchscreen (pantalla que al ser tocada efectúa los comandos del ratón en el software). Se **encuentra actualmente implementado en muchos restaurantes** y en **muchas tiendas de autoservicio de todo el mundo**. Fue iniciado **por Gene Mosher en la computadora del ST de Atari en 1986**, el uso que él **especificó en las GUI de pantalla táctil ha encabezado una revolución mundial e innovadora** en el **uso de las computadoras a través de las industrias alimenticias y de bebidas**, y en ventas al por menor.

Otros ejemplos de **GUI de uso específico, relacionados con la pantalla táctil son los cajeros automáticos**, los **kioscos de información y las pantallas de monitoreo y control en los usos industriales**, que emplean un sistema operativo de tiempo real (RTOS). Los teléfonos móviles y los sistemas o consolas de juego también emplean las pantallas táctiles. Además, **la domótica no es posible sin una buena interfaz de usuario**, o GUI.

Interfaz Natural de Usuario (NUI)

Las **NUI naturales son aquellas en las que se interactúa con un sistema**, aplicación, etcétera, **sin utilizar dispositivos de entrada como ratón, teclado, lápiz óptico**, etc. En lugar de estos **se utilizan las manos o las yemas de los dedos**. Son las interfaces **para dispositivos móviles** a las que hicimos referencia anteriormente. También se han **incorporado con poco éxito en portátiles**.

Zooming

4.3 Fundamentos de diseño de interfaz gráfica

Hasta **hace algunos años atrás, la GUI era considerada parte secundaria** al desarrollar una aplicación. Sólo **se ponía énfasis en lograr que la aplicación contara con todas las funcionalidades requeridas.** La GUI **simplemente mostraba las acciones** que se podían realizar **sin dar importancia a cómo las veía el usuario**. Con el paso del tiempo, las **aplicaciones comenzaron a formar parte de la vida cotidiana**. Cada vez más **usuarios, con o sin conocimientos, necesitaban interactuar con Interfaces de Usuario**. Justamente, **pensando en los usuarios inexpertos** se comenzó a desarrollar una **Ingeniería de Interfaces.**

La **Interfaz de Usuario es la parte del software** que las personas **pueden ver, oír, tocar, hablar; es decir, donde se pueden entende**r. La Interfaz de Usuario tiene esencialmente **dos componentes: la entrada y la salida**. La **entrada es cómo una persona le comunica sus necesidades** o deseos a la **computadora**. Algunos **componentes de entrada comunes** son el **teclado, el ratón, un dedo** (para pantallas sensibles al tacto: touch screen), y **la voz** (para las instrucciones habladas).

La salida **es la forma en que la computadora transmite los resultados a lo solicitado por el usuario**. Hoy en día el mecanismo de salida de la computadora **más común es la pantalla**, seguido de mecanismos que aprovechan las capacidades auditivas de una persona: **de voz y sonido**. En la a**ctualidad la GUI es parte fundamental de cualquier aplicación**, y por lo tanto tiene **tanta importancia como el desarrollo de la aplicación en sí.**

Modelos de GUI

Existen **tres puntos de vista distintos en una GUI: el Modelo del Usuario, el Modelo del Diseñador y el Modelo del Programador.**

- **Modelo del Usuario**: el **usuario tiene su propia visión del sistema** y espera que se **comporte de determinada manera**. El modelo de usuario se puede conocer estudiándolo a **través de tests o prototipos, entrevistas, realimentación**.
- **Modelo del Diseñador**: el diseñador **es quién se encarga de unir las ideas, necesidades y deseos del usuario**, con las **herramientas que dispone el programador para desarrollar el software**. Éste **modelo consta de tres partes**: **la Presentación** que es lo primero que **llama la atenci**ón del usuario; luego **adquiere más importancia la Interacción** que es donde el **usuario constata** si **el producto satisface sus expectativas**. La tercera y **última parte es la de Relaciones entre objetos**. Aquí **es donde se define la relación entre el modelo mental del usuario y los objetos** de la **Interfaz**.
- **Modelo del Programador**: es el modelo **más fácil de visualizar porque se puede especificar formalmente**. Este modelo consta de **los objetos que manipula el programador, distintos a los que maneja el usuario** (el programador maneja **una base de datos, el usuario la llama agenda o contactos**). El **usuario no ve los objetos que maneja el programador**. Si bien **el programador conoce de plataforma de desarrollo, sistema operativo, lenguajes y herramientas de programación**, especificaciones; no significa que **tenga la habilidad de proporcionar al usuario modelos** más **adecuados**.

Los **distintos modelos nos permiten conocer cómo visualizan la GUI** cada uno de los **'actores'**. Cada **uno de éstos son protagonistas al momento del diseño**. Conocer cada **punto de vista permite comprender los principios y reglas**. En la **siguiente sección se detallarán una serie de factores** que **deben tenerse en cuenta** junto a los principios. En el **Diseño de GUI** tal como se mencionó, el **usuario es un 'actor' importante** al momento de diseñar la GUI; por tanto, **antes de tomar decisiones** respecto al diseño también se deben **tener en cuenta las capacidades físicas y mentales** del mismo.
Existen **numerosos factores humanos que no se deben ignorar**, algunos de ellos se explican a continuación:

1. **Las personas tienen memoria limitada a corto plazo**: se pueden **recordar alrededor de siete elementos de información** (Miller, 1957). Esto significa que si **al usuario se le presenta demasiada información** al mismo tiempo seguramente **no podrá asimilarla**.
2. Todas **las personas cometen errores**, en especial cuando se trata de manejar **demasiada información o se trabaja bajo presión**. Cuando **la aplicación no responde como se espera, o emiten avisos y/o alarmas** con mensajes poco claros, **aumenta el estrés** de los usuarios.
3. Las **personas poseen un amplio rango de capacidades físicas**. Algunas ven, u oyen mejor que otras; algunas son daltónicas; otras no tienen una buena motricidad. El **diseño de la GUI no debe contemplar las propias capacidades y suponer que el resto de los usuarios** podrán **adaptarse** fácilmente.

4. **Las personas difieren en los gustos de interacción**: algunas **prefieren trabajar con menús, otras con imágenes, otras con texto**, etc. Hasta **hay personas que prefieren emitir comandos al sistema**. Cada uno de **estos factores son la base de los Principios Generales del Diseño de GUI**; por lo tanto, **deben ser considerados por el diseñador y transmitidos al programador**. Principios del Diseño Existe gran cantidad de información y bibliografía] respecto al diseño de la GUI.

Principios de diseño

En los **siguientes puntos se explicaran de manera resumida los principios más relevantes:**

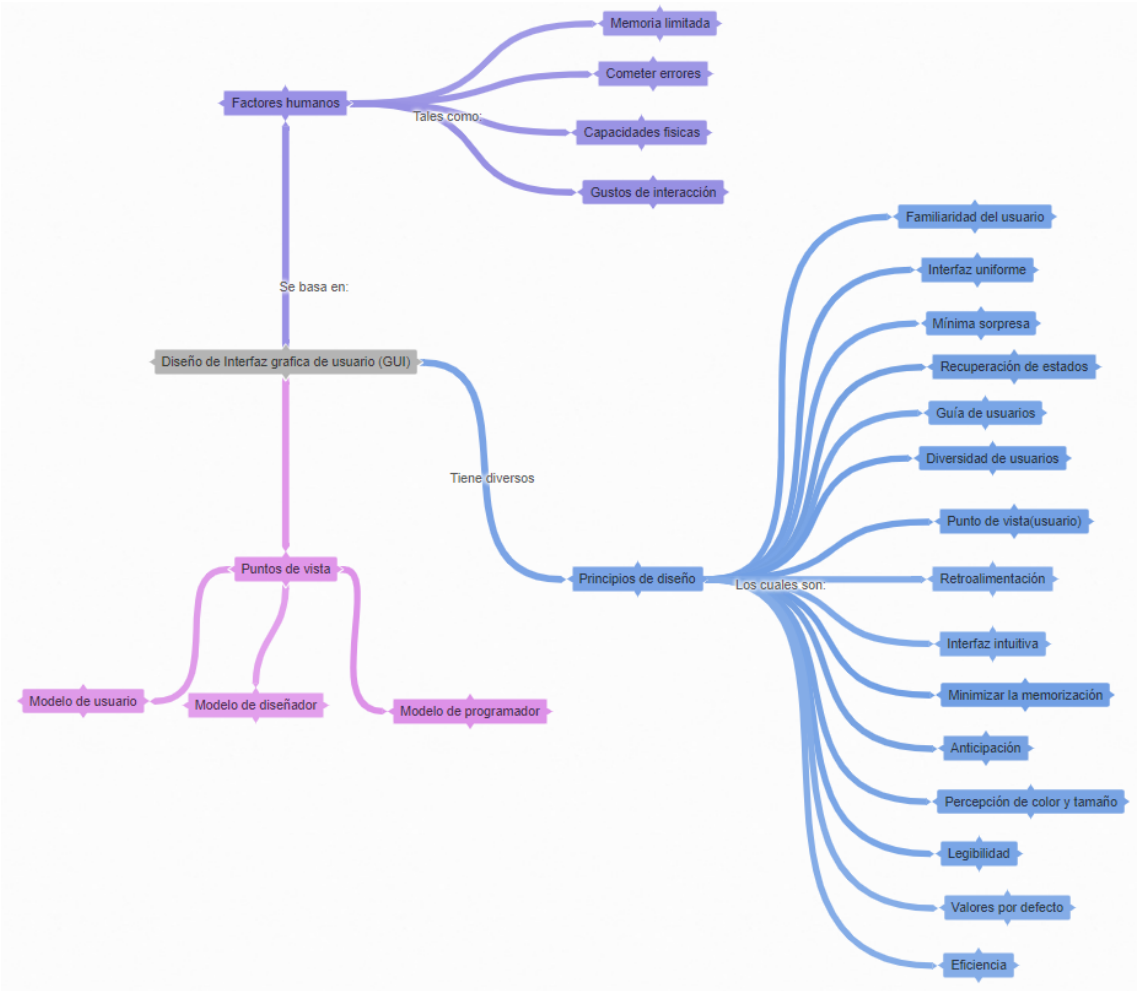
- **Familiaridad del usuario**: significa que la **interfaz debe utilizar términos e imágenes conocidos por el usuario**; y los objetos que manipula el sistema **deben estar relacionados con el ámbito de trabajo.**
- **Uniformidad de la Interfaz**: significa **que tanto comandos como menús deben tener el mismo formato**. Las **Interfaces uniformes reducen** el **tiempo de aprendizaje.**
- **Mínima sorpresa**: el **comportamiento del sistema no** debe mostrar **situaciones inesperadas**. Ante **este tipo de situaciones el usuario puede mostrar irritabilidad**, por lo **tanto perder interés** en utilizar la aplicación.
- **Recuperación de estados**: éste es **uno de los principios más importantes al diseñar una Interfaz**. Es **inevitable cometer errores**, por tanto, el sistema le **debe proporcionar al usuario la manera de subsanarlos** o volver a estados anteriores**. Este principio involucra varias acciones** como pedir **al usuario que confirme acciones destructivas**, que el usuario pueda deshacer, etc.
- **Guía de usuarios**: la Interfaz **debe proporcionar al usuario asistencia, ayuda**. No sólo cuando **se cometen errores sino también cuando no se sabe qué hacer o cómo hacer** alguna **tarea**. Esta ayuda debe estar **integrada al sistema** (algunas además ofrecen **ayuda on line**) y debe ser clara cuando el usuario la requiera, **sin saturar con información**.
- **Diversidad de usuarios:** se debe tener en cuenta **los diferentes usuarios que pueden utilizar la aplicación**. Aquellos casuales, que necesitan que los guíen, y aquellos que podrían usarla constantemente los cuales necesitarán trabajar con métodos abreviados, tan rápido como sea posible. Además, se **podría incluir recursos para mostrar diferentes tamaños de texto, reemplazar sonido por texto** y al revés, modificar **tamaño de botones,** etc. Esto refleja la noción de **Diseño Universal**, principio de diseño cuyo objetivo es **evitar excluir usuarios**.

- **Adoptar el punto de vista del usuario**: se debe **ver la interfaz desde fuera y en relación con las tareas** que va a realizar el usuario. Hay que **tener mucho cuidado** en no centrarse en los aspectos de **implementación** que hagan **perder la perspectiva**.
- **Realimentación:** la interfaz **debe dar inmediatamente alguna respuesta a cualquier acción del usuario**. Por ejemplo: **movimiento del cursor, resaltar la opción elegida de un menú**, comunicar **el éxito o fracaso** de una tarea. La completitud de tareas o los errores deben ser informados.

- **Potenciar la sensación de control** del usuario sobre el sistema, **especialmente para los usuarios sin experiencia**: que la interfaz **sea intuitiva (utilizar iconos, modelos,** métodos, etc. consistentes con otras aplicaciones y con el mundo real), **facilitar la exploración** (todas las operaciones deben ser accesibles desde el menú principal), permitir **cancelar y deshacer** operaciones, etc.
- **Minimizar la necesidad de memorización**: usar **controles gráficos, limitar la carga de información a corto plazo**, procurar que la **información necesaria en cada momento** esté presente en la pantalla, **utilizar nombres y símbolos auto-explicativos** y fáciles de recordar, etc.
- **Anticipación:** la **aplicación debe anticiparse a las necesidades del usuario**, y no esperar a que tenga que buscar información.
- **Percepción de color y tamaño:** se debe tener en cuenta a aquellos **usuarios con problema de visualización del color, pero es muy útil usar convención de colores**. Además, al **mostrar varios objetos en la pantalla deben estar distribuidos**, debe haber distancia entre ellos para que así el usuario **pueda percibirlos sin sobrecarga.**
- **Legibilidad: no sólo se debe prestar atención a los colores y a los objetos que se ven en pantalla** sino también a **cómo se verá el texto**. El **tipo y tamaño de letra debe ser legible**, y el color debe contrastar con el fondo (utilizar letras negras en fondo claro).
- **Valores por defecto**: lo ideal **es utilizar 'valores estándar'**. Se debe **tener en cuenta que los valores por defecto** deben ser opciones **inteligentes, sensatas y fáciles** de modificar. •
- **Eficiencia**: se **debe considerar la productividad como ideal a lograr**. **El usuario no debe esperar la respuesta del sistema** por tiempo prolongado; los mensajes de a**yuda, menús y etiquetas deben ser sencillos** y deben **utilizar palabras claves** para poder transmitir fácilmente a qué hacen referencia.

Cada **uno de estos principios ayuda a hacer un buen diseño de la**

GUI. Cabe aclarar que éste es un resumen de los más relevantes, no es el **propósito de esta asignatura el diseño de interfaces.**



4.4 Nuevas Tendencias en el diseño de interfaces gráficas y diseño gráfico.

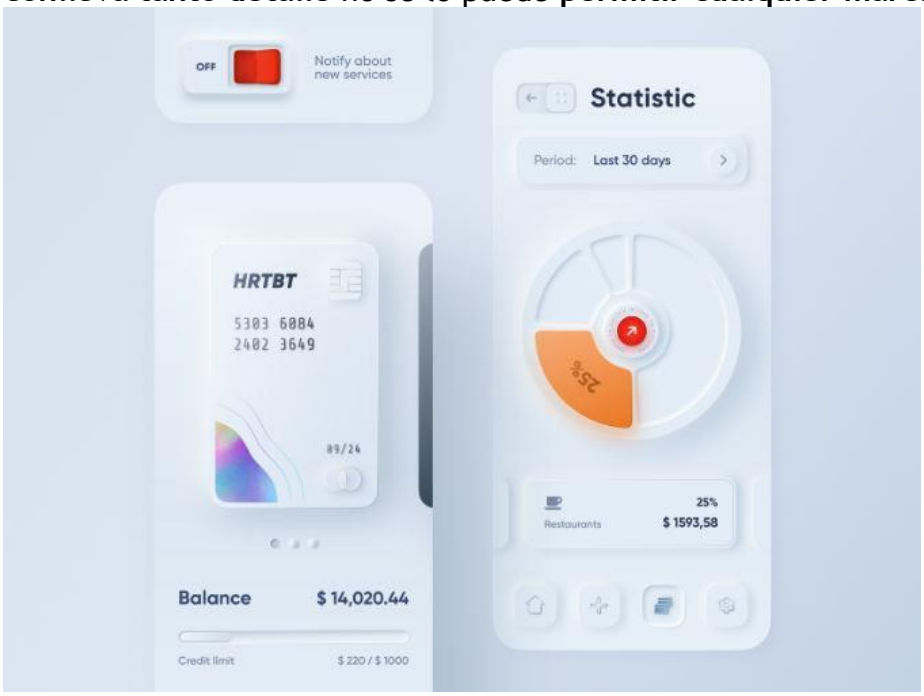
Vamos a **repasar las tendencias actuales en el diseño de interfaces** y haremos

referencia a un sitio web desarrollado por Google para la creación de interfaces gráficas para móvil y web que me parece de máxima utilidad.

Neumorfismo / Skeuomorfismo

El neumorfismo **quizá es la gran nueva tendencia de este 2020** en el diseño de interfaz de usuario o al **menos la más rompedora.** Todo empezó cuando un **usuario de Dribbble llamado alexplyuto subió un diseño a finales de 2019** que consiguió más de 3000 likes y a **partir de ahí conceptos similares empezaron a aparecer**.

Este **estilo se ve representado por un diseño muy detallado y perfeccionista**, detalla **sombras, brillos y degradados**. Lo vamos a ver en **diseños para grandes compañías** que ofrecen cuentas de usuario, ya que **el trabajo de diseño que conlleva tanto detalle** no se lo puede **permitir cualquier marca.**



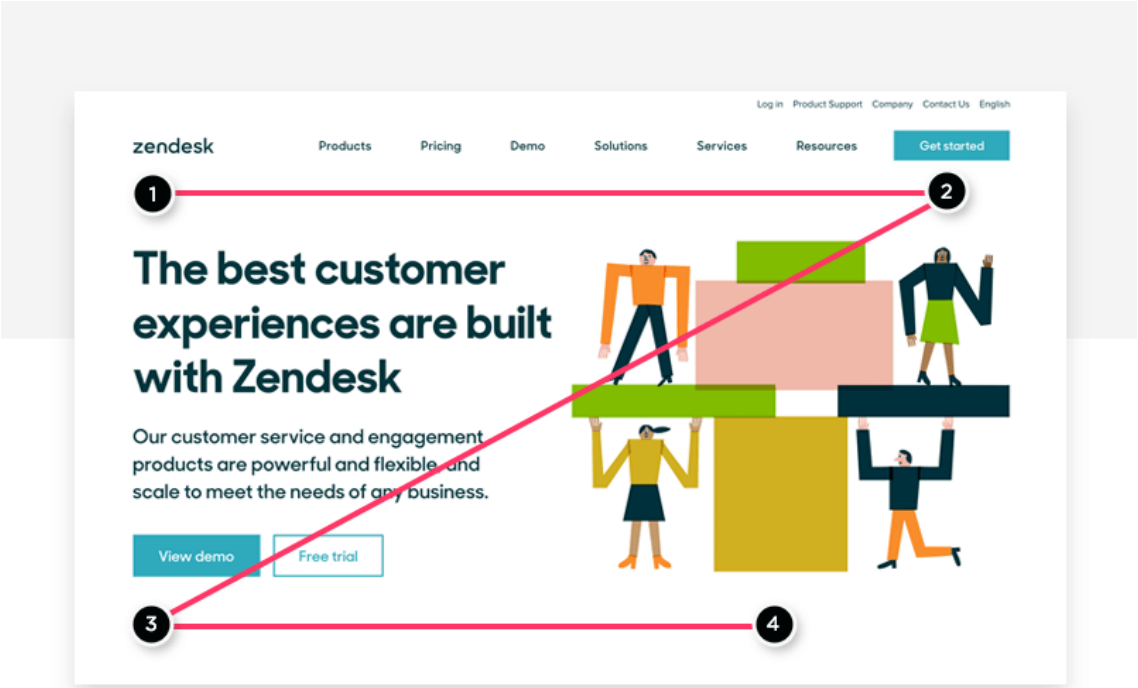
Después de **tantos años de flat design y simplismo** en el diseño **volvemos al realismo**, eso sí, está vez es **realismo futurista**, ya **veremos cómo aguanta el paso del tiempo** pero al menos para este 2020 es la **tendencia más transgresora.**

Storytelling

El **storytelling o cuentacuentos en castellano, empieza a estar muy de moda en el mundo del diseño web**, pero no es cosa de ahora. Se **comenzó Flash de Macromedia (posteriormente adquirido por Adobe) ahora en el olvido**. A través de su **programación vía timeline, te permitía crear historias y hacer interactuar al usuario** al estilo de los libros **"Elige tu propia aventura".**

La **narración de historias puede ayudar a la interacción entre producto digital y el usuario**, creando **emociones positivas y haciendo que el receptor guarde un buen y memorable recuerdo** de la **marca**. Por lo tanto, el **storytelling es una genial herramienta de marketing digital que** puede aumentar considerablemente las ventas de un producto o servicio.

El **storytelling es tendencia en diseño UI para el 2020** y cada vez lo será más. Aquí podéis ver algunos ejemplos:



Realidad Aumentada

Poco **a poco la realidad aumentada va teniendo más implicación en el mundo del diseño** de **interfaz web y app**, ya se están viendo algunos **casos desarrollados para grandes marcas,** como el de **la APP IKEA** Place que **te permite escanear una habitación con tu smartphone e interactuar** con el **mobiliario 3D**.

https://youtu.be/UudV1VdFtuQ

Hemos e**mpezado con filtros de Snapchat o Instagram pero a esta tecnología le queda un largo recorrido, Apple y Google** lo saben perfectamente y por **ello ya han presentado sus propias plataformas de desarrollo de realidad aumentada:**

ARKit (Apple)
ARCore (Google)

Este es **un indicativo que la realidad aumentada ya es una realidad factible**, que nos presta **aplicaciones de uso habitual**, por eso está en **esta lista de tendencias en diseño UI** para el 2020.

Este mundo no para y las nuevas tecnologías nos obligan a estar preparados para aprender nuevas herramientas, seguro que **en los próximos años la capacidad de crear interfaces de realidad aumentada y elementos 3D** será muy **valorada.**

Diseños Asimétricos

En los últimos tiempos **las estructuras de contenido de las páginas web se han basado en cuadrículas**, donde es fácil **decirle al usuario que debe leer y en que orden**, todo lo contrario que en **el asimetrismo**, **tendencia en diseño UI para el 2020**.

Es difícil crear un **diseño asimétrico y no despistar al usuario con mucha foto flotando y letra suelta** por la pantalla, pero si **se hace con delicadeza** se puede llegar a diseñar un sitio muy **interesante**.

Para **las grandes marcas es complicado** ya que **se la juegan con el riesgo de crear una experiencia web caótica** para sus usuarios, pero existen sitios personales que están llevando esta idea a **niveles muy imaginativos**.

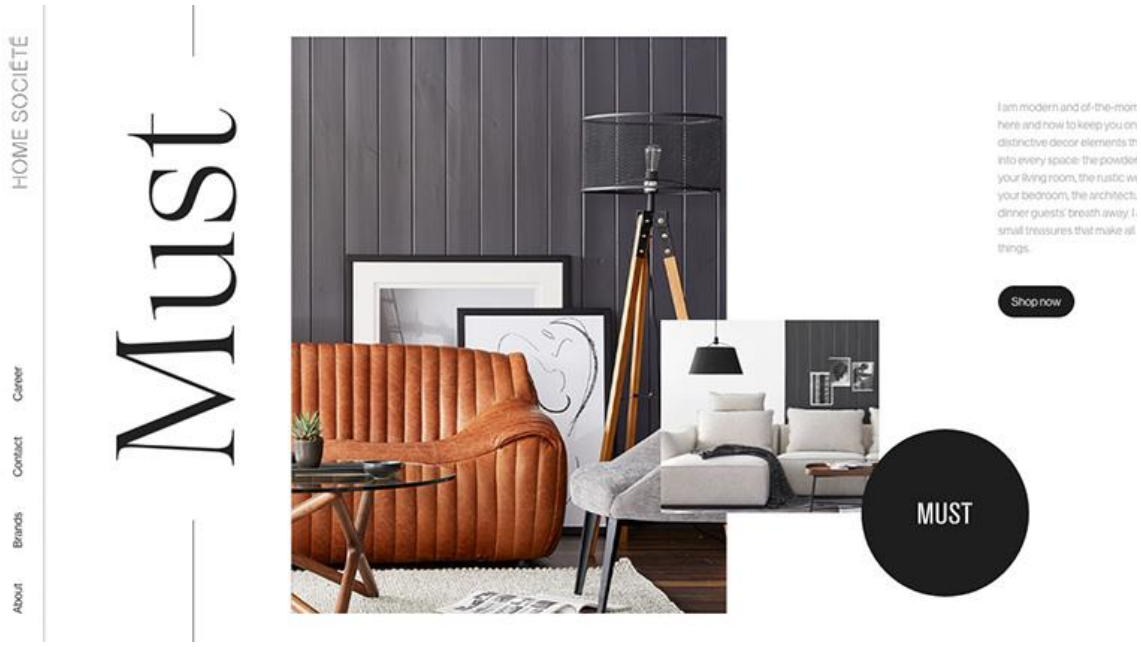
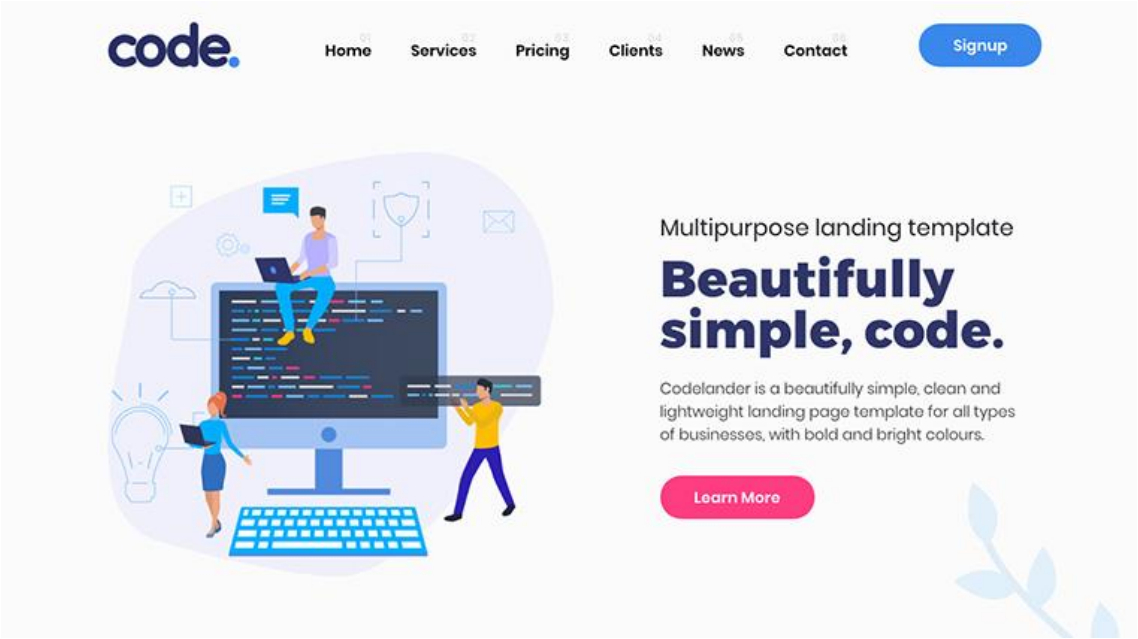


Ilustración y animación

Durante el 2019 se **ha visto bastante ilustración en el diseño de interfaz de páginas web y apps**, el estilo **más de moda ha sido y sigue siendo la ilustración simplificada**, colores **planos de tono alegre y dinámico**, mucho **personaje interactuando con dispositivos u otros objetos**. Hay **gran material en internet para que los diseñadores web** que **no dominan de ilustración** puedan **usar como recurso y un ejemplo de ello es undraw,** un banco de **ilustraciones de colores personalizables** creado por la **griega Katerina Limpitsouni.**



Para **el 2020 los estilos de ilustración** que se implementan **en sitios web o apps se reinventan, si como yo ya empiezas a estar un poco harto/a** de tanta **ilustración** de archivo, este año ya estamos viendo o**tros estilos como el 3D, diseño absurdo, abstracto, futurístico, isométrico…** Cualquiera de estos **estilos en un diseño pueden decorar y dar un toque orgánico** a la interfaz de usuario.

5 Java AWT y Swing

5.1 INTRODUCCIÓN

La *interfaz de usuario* es la parte del programa que permite a éste interactuar con el usuario. Las interfaces de usuario pueden adoptar muchas formas, que van desde la simple línea de comandos hasta las interfaces gráficas (IGU-GUI) que proporcionan las aplicaciones más modernas.
La interfaz de usuario es uno de los aspectos más importante de cualquier aplicación. Una aplicación sin un interfaz fácil, impide que los usuarios saquen el máximo rendimiento del programa. Java proporciona los elementos básicos para construir interfaces de usuario a través de la biblioteca de clases **AWT**, y opciones para mejorarlas mediante una nueva biblioteca denominada **Swing.**
Debido a que el lenguaje de programación Java es **independiente de la plataforma** en que se ejecuten sus aplicaciones, la biblioteca AWT también es independiente de la plataforma en que se ejecute. El AWT proporciona un conjunto de herramientas para la construcción de interfaces gráficas que tienen una apariencia y se comportan de forma semejante en todas las plataformas en que se ejecute.
Los elementos de la interfaz proporcionados por la biblioteca AWT están implementados utilizando ***toolkits*** nativos de las plataformas, preservando una apariencia semejante a todas las aplicaciones que se creen para esa

plataforma. Este es un punto fuerte del AWT, pero también tiene la desventaja de que un interfaz gráfico diseñado para una plataforma, puede no visualizarse correctamente en otra diferente. Estas carencias del AWT son subsanadas en parte por **Swing**, y en general por las **JFC (Java Foundation Classes).**

5.2 AWT (ABSTRACT WINDOW TOOLKIT)

AWT es el acrónimo del Abstract Window Toolkit para Java. Se trata de una biblioteca de clases Java para el desarrollo de las Interfaces de Usuario Gráficas. La versión del AWT que *Sun* proporciona con el JDK 1.0.x se desarrolló en sólo dos meses y es la parte más débil de todo lo que representa Java como lenguaje. El entorno que ofrece es demasiado simple, no se han tenido en cuenta las ideas de entornos gráficos novedosos. Quizá la presión de tener que lanzar algo al mercado haya tenido mucho que ver en la pobreza de AWT en la versión 1.0.x.

Javasoft, en vista de la precariedad de que hace gala el AWT, y para asegurarse que los elementos que desarrolla para generar interfaces gráficas sean fácilmente transportables entre plataformas, se ha unido con Netscape, IBM y Lighthouse Design para crear un conjunto de clases que proporcionen una sensación visual agradable y sean más fáciles de utilizar por el programador. Esta colección de clases son las ***Java Foundation Classes*** (**JFC**), que están constituidas por cinco grupos de clases, al menos en este momento: AWT, Java 2D, Accesibilidad, Arrastrar y Soltar y Swing.

- **AWT** engloba a todos los componentes del AWT que existían en la versión 1.1.2 del JDK y en los que se han incorporado en versiones posteriores.
- **Java 2D** es un conjunto de clases gráficas bajo licencia de IBM/Taligent, que todavía está en construcción.
- **Accesibilidad**, proporciona clases para facilitar el uso de ordenadores y tecnología informática a disminuidos, tiene lupas de pantalla, y cosas así.
- **Arrastrar y Soltar** (*Drag and Drop*), son clases en las que se soporta Glasgow, que es la nueva generación de los JavaBeans.
- **Swing**, es la parte más importante y la que más desarrollada se encuentra. Ha sido creada en conjunción con Netscape y proporciona una serie de componentes muy bien descritos y especificados de forma que su presentación visual es independiente de la plataforma en que se ejecute el applet o la aplicación que utilice estas clases. **Swing** simplemente extiende el AWT añadiendo un conjunto de componentes, ***JComponents***, y sus clases de soporte. Hay un conjunto de componentes de Swing que son análogos a los de AWT, y algunos de ellos participan de la arquitectura MVC (*Modelo-Vista-Controlador*), aunque **Swing** también proporciona otros widgets nuevos como árboles, pestañas, etc.

  Se recomienda encarecidamente a la hora de programar el uso de Swing en

lugar de AWT ya que ésta última versión soluciona múltiples problemas existentes en la versión predecesora (AWT). Sin embargo, dado que a la hora de programar las diferencias entre Swing y AWT son prácticamente nulas, en este tema seguiremos basándonos en la librería AWT para dar todas las explicaciones (dado que fue este el modelo de programación visual que surgió originalmente). Posteriormente (dentro de la explicaciones de clase) se indicarán las modificaciones que hay que hacer sobre una programa en AWT para convertirlo a un programa en Swing.

5.3 ESTRUCTURA BÁSICA DE AWT

La estructura básica del AWT se basa en:

- Componentes
- Contenedores

Los *Componentes* son los controles básicos como botones, listas, cuadros de texto, checkbox,...
Los *Contenedores* contienen a otros *Componentes* dentro, los cuales de encuentran posicionados de forma relativa con respecto al contenedor. No se usan posiciones fijas de los Componentes con respecto a los Contenedores, sino que están situados a través de una disposición controlada (*layouts*). Los Contenedores son a su vez también Componentes.
La interacción con el usuario y el sistema se realiza por medio de eventos. Así el manejo de eventos se encarga de tratar la interacción del usuario con el ratón, menús, etc. Los eventos pueden tratarse tanto en Contenedores como en Componentes, corriendo por cuenta del programador la seguridad de tratamiento de los eventos adecuados. La gestión de eventos se trata en el

Con Swing se va un paso más allá, ya que todos los *JComponentes* son subclases de **Container**, lo que hace posible que widgets Swing puedan contener otros componentes, tanto de AWT como de Swing.

5.4 Componentes y Contenedores

La interfaz gráfica de usuario (IGU) está construida en base a elementos gráficos básicos denominados ***Componentes***. Típicos ejemplos de estos Componentes son los botones, barras de desplazamiento, etiquetas, listas, cajas de selección o campos de texto.

Los *Componentes* permiten al usuario interactuar con la aplicación. En la biblioteca AWT, todos los Componentes de la interfaz gráfica de usuario son instancias de la clase ***Component*** o de una clase descendiente de ella.

Los Componentes no se encuentran aislados, sino agrupados dentro de *Contenedores*. Los ***Contenedores*** contienen y organizan la situación de los Componentes. Los Contenedores son en sí mismos Componentes y como tales pueden ser situados dentro de otros Contenedores. En la biblioteca AWT, todos los Contenedores son instancias de la clase ***Container*** o una clase descendiente de ella.

5.5 PRIMEROS PASOS CON AWT: CREACIÓN DE FRAMES. Practica guiada frames

Un frame (marco) es una ventana que no está contenida dentro de otra ventana. Es un contenedor (container).
Los contenedores pueden almacenar otros elementos del interfaz de usuario denominados componentes (Components), por ejemplo botones o campos de texto.
Ejemplo 1: Primer Frame.

Construcción de un Frame en blanco.

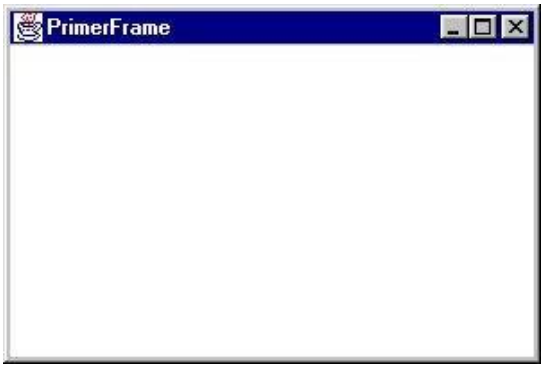


Figura 7.1: Frame en blanco

Código:

```
import java.awt.*;
/* Esta clase hereda de la clase Frame de Java, es la
* clase que va a representar un Frame.  */ public class MiFrame extends Frame
{
                public MiFrame()
                {
                        setSize(300, 200);
                  setTitle("PrimerFrame");
                 }

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

```
                        // Creamos una instancia de esta clase
  MiFrame f =  new  MiFrame();     // Mostramos el frame     f.show();
                  }
}
```

Jerarquía de la clase *Frame* dentro de la AWT.

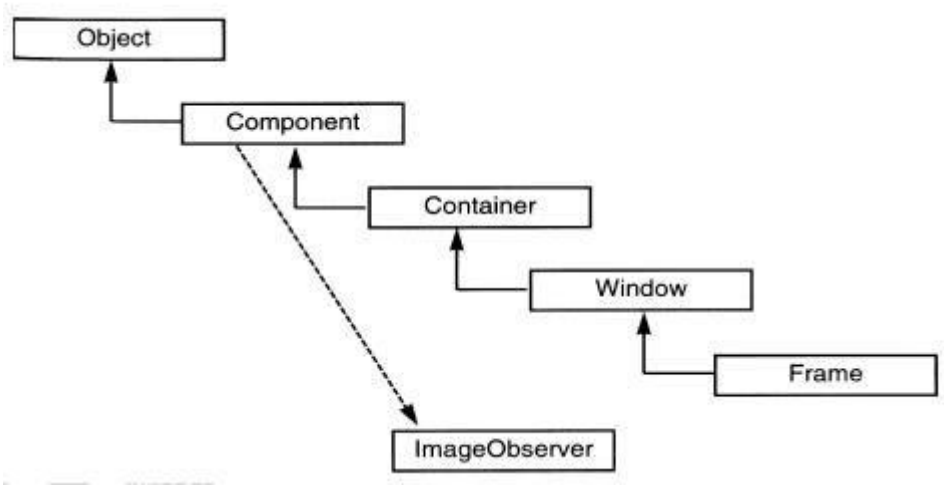


Figura 7.2: Jerarquía de la clase Frame

6 COMPONENTES

Component es una clase abstracta que representa todo lo que tiene una posición, un tamaño, puede ser pintado en pantalla y puede recibir eventos.

Los Objetos derivados de la clase **Component** que se incluyen en el Abstract Window Toolkit son los que aparecen a continuación:

- Button
- Canvas
- Checkbox
- Choice
- Container
    - Panel
    - Window
    - Dialog
    - Frame
- Label
- List
- Scrollbar

- TextComponent
  - TextArea
  - TextField

Vamos a ver un poco más en detalle los Componentes que nos proporciona el AWT para incorporar a la creación de la interface con el usuario.

6.1 Botones. Practica guiada botones

Veremos ejemplos de cómo se añaden botones a un panel para la interacción del usuario con la aplicación, pero antes vamos a ver la creación de botones como objetos.

Se pueden crear objetos Button con el operador **new**:

```
Button boton;
boton = new Button( "Botón");
```

La cadena utilizada en la creación del botón aparecerá en el botón cuando se visualice en pantalla. Esta cadena también se devolverá para utilizarla como identificación del botón cuando ocurra un evento.

6.1.1 Botones de Pulsación

Los botones presentados en la ventana son los botones de pulsación estándar; no obstante, para variar la representación en pantalla y para conseguir una interfaz más limpia, AWT ofrece a los programadores otros tipos de botones.

6.1.2 Botones de Lista

Los botones de selección en una lista (*Choice*) permiten el rápido acceso a una lista de elementos.

Por ejemplo, podríamos implementar una selección de colores y mantenerla en un botón Choice:

Figura 7.3: Lista desplegable

Código:

```
import java.awt.*;
```

```
public class PanelBotonSeleccion extends Panel{
                Choice Selector;

                public PanelBotonSeleccion(){
                        Selector = new Choice();

                        Selector.addItem("Rojo");
                        Selector.addItem("Verde");
                        Selector.addItem("Azul");

                        add(Selector);
                }

                public static void main(String[] args){
                        // No intentar comprender este código por el momento
                        Frame f = new Frame();
                        f.add(new PanelBotonSeleccion());
                        f.show();
                }
}
```

6.1.3 Botones de Marcación

Los botones de marcación (*Checkbox*) se utilizan frecuentemente como botones de estado. Proporcionan información del tipo *Sí* o *No* (true o false).

Figura 7.4: Botón de Selección Múltiple (CheckBox)

Código:

```
import java.awt.*;

public class PanelBotonComprobacion extends Panel{
```

```
                Checkbox Relleno;


        public PanelBotonComprobacion(){                    Relleno = new Check
box ("Relleno");


                        add(Relleno);
                }


                public static void main(String[] args){
                        // No intentar comprender este código por el momento
                        Frame f = new Frame();
                        f.add(new PanelBotonComprobacion());
                        f.show();
                }
}
```

6.1.4 Botones de Selección

Los botones de selección se pueden agrupar para formar una interfaz de botón de radio (*CheckboxGroup*), que son agrupaciones de botones Checkbox en las que siempre hay un único botón activo.

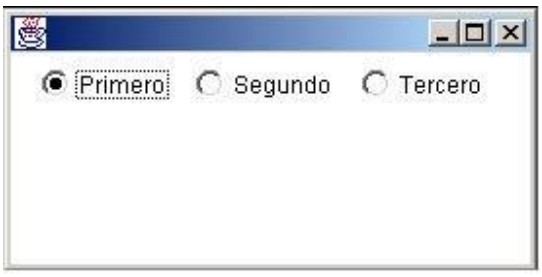


Figura 7.5: Botones de selección exclusiva

Código:

```
import java.awt.*;
 public class PanelBotonRadio extends Panel{
                CheckboxGroup Radio;


        public PanelBotonRadio(){                Radio = new CheckboxGroup()
;       add( new Checkbox( "Primero", Radio, true) );            add( new Ch
eckbox( "Segundo", Radio, false) );              add( new Checkbox( "Tercero
", Radio, false) );
```

```
                }

                public static void main(String[] args){
                        // No intentar comprender este código por el momento
                        Frame f = new Frame();
                        f.add(new PanelBotonRadio());
                        f.show();
                }
}
```

En el ejemplo anterior, observamos que siempre hay un botón activo entre los que conforman el interfaz de comprobación que se ha implementado.

6.2 Etiquetas. Practica guiada etiquetas

Las etiquetas (*Label*) proporcionan una forma de colocar texto estático en un panel, para mostrar información que no varía, normalmente, al usuario.

Este ejemplo presenta dos textos en pantalla, tal como aparece en la figura siguiente:

Figura 7.6: Dos etiquetas

Código:

```
import java.awt.*;

public class PanelEtiquetas extends Panel{

                public PanelEtiquetas(){
                Label etiq1 = new Label( "Hola java!" );
                Label etiq2 = new Label( "Otra etiqueta" );

        add(etiq1);             add(etiq2);
                }

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

```
                        // No intentar comprender este código por el momento
                        Frame f = new Frame();
                f.add(new PanelEtiquetas());
                f.show();
}
}
```

6.3 Listas. Practica guiada listas

Las listas (*List*) aparecen en los interfaces de usuario para facilitar a los operadores la manipulación de muchos elementos. Se crean utilizando métodos similares a los de los botones Choice. La lista es visible todo el tiempo, utilizándose una barra de desplazamiento para visualizar los elementos que no caben en el área que aparece en la pantalla.

El ejemplo siguiente, crea una lista que muestra cuatro líneas a la vez y no permite selección múltiple.

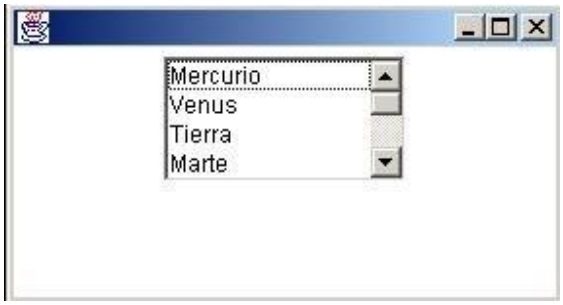


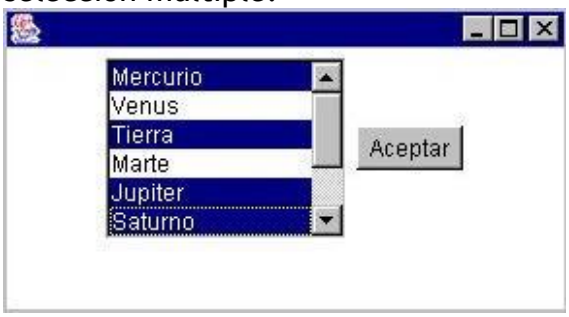
Figura 7.7: Lista de Selección Exclusiva

Código:

```
import java.awt.*;

public class PanelLista extends Panel{

                public PanelLista(){
                List l = new List( 4, false );

                        l.addItem( "Mercurio" );
                l.addItem( "Venus" );
                l.addItem( "Tierra" );
                l.addItem( "Marte" );
                l.addItem( "Jupiter" );
                l.addItem( "Saturno" );
```

```
                l.addItem( "Neptuno" );
                l.addItem( "Urano" );
        l.addItem( "Plutón" );
                add( l );
    }
                public static void main(String[] args){
                // No intentar comprender este código por el momento
                Frame f = new Frame();
                f.add(new PanelLista());
                        f.show();
                }
}
```

En la aplicación siguiente, se permite al usuario seleccionar varios elementos de los que constituyen la lista. En la figura se muestra la apariencia de una selección múltiple.



Código:

```
 import java.awt.*;


public class PanelListaMult extends Panel{


                List lm = new List(6,true);


                public PanelListaMult(){


                        Button boton = new Button("Aceptar");


                lm.addItem("Mercurio");                  lm.addItem("Venus")
;               lm.addItem("Tierra");            lm.addItem("Marte");
        lm.addItem("Jupiter");           lm.addItem("Saturno");           lm
.addItem("Neptuno");             lm.addItem("Urano");

                        lm.addItem("Pluton");
```

```
                        add(lm);
                        add(boton);
                }

                public static void main(String[] args){
                        // No intentar comprender este código por el momento
                Frame f = new Frame();
                f.add(new PanelListaMult());
                f.show();
                }
}
```

6.4 Campos de Texto. Practica guiada campos de texto

Para la entrada directa de datos se suelen utilizar los campos de texto, que aparecen en pantalla como pequeñas cajas que permiten al usuario la entrada por teclado.

Figura 7.9: Campos de Texto

Los campos de texto (*TextField*) se pueden crear vacíos, vacíos con una longitud determinada, rellenos con texto predefinido y rellenos con texto predefinido y una longitud determinada. Este ejemplo genera cuatro campos de texto con cada una de las características anteriores. La imagen muestra los distintos tipos de campos.

Código:

```
import java.awt.*;
 public class PanelCamposTexto extends Panel {

                TextField tf1,tf2,tf3,tf4;

                public PanelCamposTexto(){
```

```
        //Campo de texto vacío           tf1 = new TextField();

        //Campo de texto vacío con 20 columnas            tf2 = new TextField
( 20 );         //Texto predefinido              tf3= new TextField( "Hola" )
;       //Texto predefinido en 30 columnas       tf4= new TextField( "Hola",
30);



        add (tf1);               add (tf2);               add          (tf3);
        add (tf4);

                }


                public static void main(String[] args){

                        // No intentar comprender este código por el momento

                        Frame f = new Frame();

                        f.add(new PanelCamposTexto());

                        f.show();

                }
}
```

6.5 Áreas de Texto

Java, a través del AWT, permite incorporar texto multilínea dentro de zonas de texto (*TextArea*). Los objetos TextArea se utilizan para elementos de texto que ocupan más de una línea, como puede ser la presentación tanto de texto editable como de sólo lectura.
Para crear un área de texto se pueden utilizar cuatro formas análogas a las que se han descrito en la creación de Campos de Texto. Pero además, para las áreas de texto hay que especificar el número de columnas.

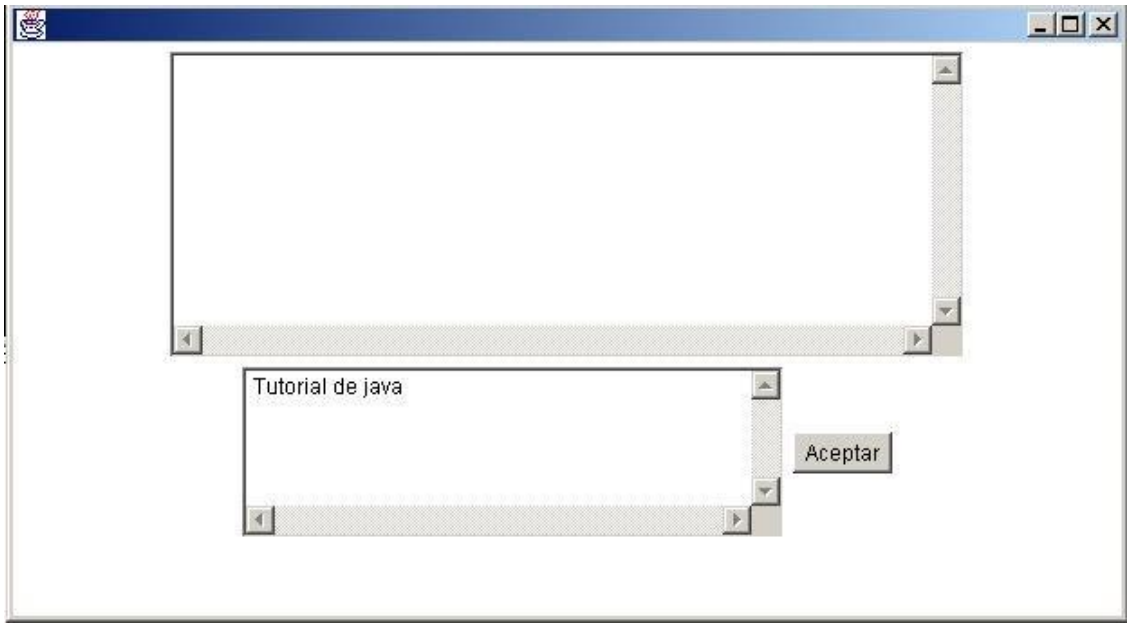


Figura 7.10: Dos Área de Texto

Se puede permitir que el usuario edite el texto con el método *setEditable()* de la misma forma que se hacía en el TextField. En la figura aparece la epresentación de AreaTexto.java, que presenta dos áreas de texto, una vacía y editable y otra con un texto predefinido y no editable.

Código:

```
import java.awt.*;


public class PanelAreaTexto extends Panel{
                TextArea t1,t2;


                public PanelAreaTexto(){
                Button boton = new Button ( "Aceptar" );          t1 = new Te
xtArea();
        t2 = new TextArea( "Tutorial de java", 5 , 40 );          t2.setEdit
able(false);


        add( t1 );              add( t2 );
                add( boton );
                }


                public static void main(String[] args){
                        // No intentar comprender este código por el momento
```

```
                        Frame f = new Frame();
                        f.add(new PanelAreaTexto());
                        f.show();
                }
}
```

Para acceder al texto actual de una zona de texto se utiliza el método *getText()*.

7 LAYOUTS. Practica guiada layouts

Los *layout managers* o *manejadores de composición*, en traducción literal, permiten controlar la disposición de los diversos componentes dentro de un contenedor. Es decir, especifican la distribución que tendrán los Componentes a la hora de colocarlos sobre un Contenedor. Java dispone de varios, en la actual versión, tal como se muestra en la imagen:

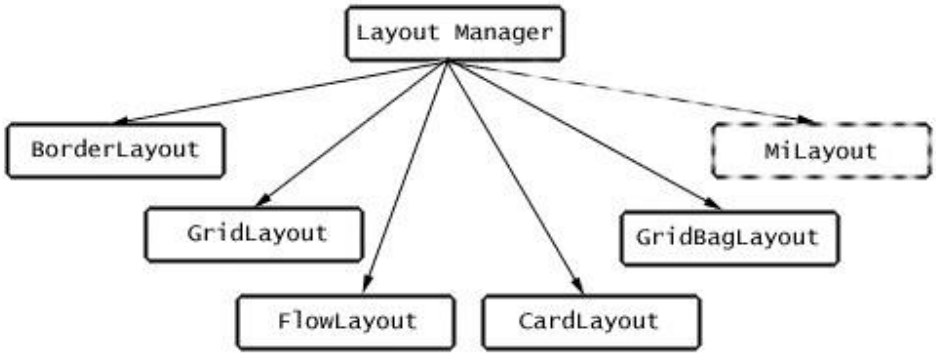


Figura 7.16: Clasificación de Layouts

¿Por qué Java proporciona estos esquemas predefinidos de disposición de componentes? La razón es simple: imaginemos que deseamos agrupar objetos de distinto tamaño en celdas de una rejilla virtual: si confiados en nuestro conocimiento de un sistema gráfico determinado, y codificamos a mano tal disposición, deberemos prever el redimensionamiento de la ventana, su repintado cuando sea cubierto por otra ventana, etc., además de todas las cuestiones relacionadas con un posible cambio de plataforma (uno nunca sabe a donde van a ir a parar nuestras ventanas).
Sigamos imaginando, ahora, que un hábil equipo de desarrollo ha previsto las disposiciones gráficas más usadas y ha creado un gestor para cada una de tales configuraciones, que se ocupará, de forma transparente para nosotros, de todas esas cuitas de formatos. Bien, pues estos gestores son instancias de las distintas clases derivadas de Layout Manager y que se utilizan en los contenedores de nuestras aplicaciones.
Los Layouts liberan al programador de tener que preocuparse de dónde ubicar cada uno de los componentes cuando una ventana es redimensionada o cuando una ventana es refrescada o cuando una ventana es llevada a una plataforma

que maneja un sistema de coordenadas diferente.

7.1 FlowLayout

Es el más simple y el que se utiliza por defecto en todos los Paneles si no se fuerza el uso de alguno de los otros. Los Componentes añadidos a un Panel con FlowLayout se encadenan en forma de lista. La cadena es horizontal, de izquierda a derecha, y se puede seleccionar el espaciado entre cada Componente.

Por ejemplo, podemos poner un grupo de botones con la composición por defecto que proporciona FlowLayout:

Código:

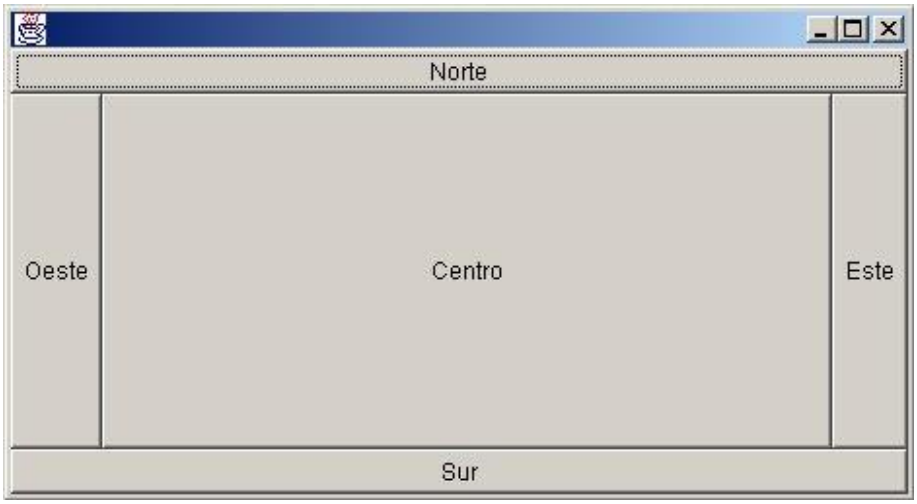
```
 import java.awt.*;


public class PanelAwtFlow extends Panel{

                Button boton1,boton2,boton3;


                public PanelAwtFlow(){
                         boton1 = new Button ( "Aceptar" );
                boton2 = new Button ("Salir");
                boton3 = new Button ("Cerrar");


                add(boton);
        add(boton2);             add(boton3);
                }


                public static void main(String[] args){
                         // No intentar comprender este código por el momento
                         Frame f = new Frame();
```

```
                        f.add(new PanelAwtFlow());
                        f.show();
                }
}
```

Este código, construye tres botones con un pequeño espacio de separación entre ellos. No se especifica ningún layout porque se usa el layout por defecto de los paneles que es el FlowLayout.

7.2 BorderLayout

La composición BorderLayout (de borde) proporciona un esquema más complejo de colocación de los Componentes en un panel. La composición utiliza cinco zonas para colocar los Componentes sobre ellas: Norte, Sur, Este, Oeste y Centro. Es el layout o composición que se utilizan por defecto Frame y Dialog. El Norte ocupa la parte superior del panel, el Este ocupa el lado derecho, Sur la zona inferior y Oeste el lado izquierdo. Centro representa el resto que queda, una vez que se hayan rellenado las otras cuatro partes.



Con BorderLayout se podrían representar botones de dirección:

```
import java.awt.*;
 public class PanelAwtBord extends Panel{
   Button botonN,botonS,botonE,botonO,botonC;

                public PanelAwtBord(){

                setLayout (new BorderLayout() );
```

```
                botonN = new Button ( "Norte" );          botonS = new Button
("Sur");        botonE = new Button ("Este");             botonO = new Button
( "Oeste" );             botonC = new Button ( "Centro" );


             add( "North", botonN );              add(   "South",   botonS   );
        add( "East", botonE);            add( "West", botonO);

                add( "Center", botonC);

                }


                public static void main(String[] args){

                        // No intentar comprender este código por el momento

                        Frame f = new Frame();

                        f.add(new PanelAwtBord());

                        f.show();

                }
}
```

Como se puede observar en el código, la manera de asignar un determinado layout a un contenedor es mediante el uso del método *setLayout ()* invocado sobre el propio contenedor:

```
<Objeto_Contenedor>.setLayout(<Objeto_Layout>)
```

Una vez asignado el layout al contenedor, la manera de añadir elementos a ese contenedor será mediante el método *add()* y en este caso indicando dónde queremos que se sitúe el elemento (North, South, East, West, Center).

7.3 GridLayout

La composición GridLayout proporciona gran flexibilidad para situar Componentes. El layout se crea con un número de filas y columnas y los Componentes van dentro de las celdas de la tabla así definida.

En la figura siguiente se muestra un panel que usa este tipo de composición para posicionar seis botones en su interior, con tres filas y dos columnas que crearán las seis celdas necesarias para albergar los botones:

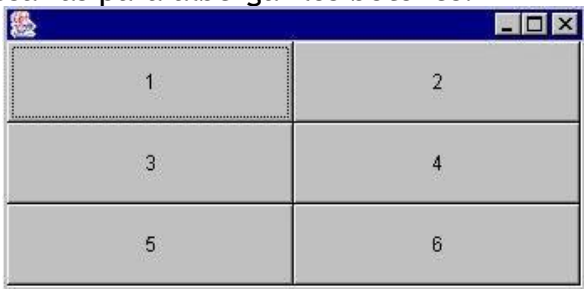


Figura 7.19: Ejemplo

GridLayout Código:

```
import java.awt.*;


public class PanelAwtGrid extends Panel{


                Button boton1,boton2,boton3,boton4,boton5,boton6;


                public PanelAwtGrid(){


                setLayout (new GridLayout( 3,2 ) );


                boton1 = new Button ( "1" );              boton2 = new Button
("2");          boton3 = new Button ("3");
        boton4 = new Button ( "4" );              boton5 = new Button ( "5" )
;               boton6 = new Button ("6");


        add( boton1 );          add( boton2 );          add(    boton3    );
        add( boton4 );          add( boton5 );          add( boton6 );
                }


                public static void main(String[] args){
                        // No intentar comprender este código por el momento
                        Frame f = new Frame();
                        f.add(new PanelAwtGrid());
                        f.show();


                }
```

```
        }
```

7.4 CardLayout

Este es el tipo de composición que se utiliza cuando se necesita una zona de la ventana que permita colocar distintos Componentes en cada momento. Este layout suele ir asociado con botones de lista (Choice), de tal modo que cada selección determina el panel (grupo de componentes) que se presentarán.
En la figura siguiente mostramos el efecto de la selección sobre la apariencia de la ventana que contiene el panel con la composición CardLayout:


```
import java.awt.*;


public class PanelAwtCard extends Panel{   Panel card;

  final static String PanelBoton = "Panel con botones";   final static String
PanelTexto = "Panel con campo de texto";



                public PanelAwtCard(){


                setLayout (new BorderLayout() );


                        Panel ac = new Panel();

                Choice c = new Choice();


                c.addItem(PanelBoton);

        c.addItem(PanelTexto);          ac.add(c);              add("North
", ac);


        card = new Panel();             card.setLayout( new CardLayout() );


        Panel p1 = new Panel();         p1.add(  new  Button  ("Botón  1")  );
        p1.add( new Button ("Botón 2") );       p1.add( new Button ("Boton 3
") );          Panel p2 = new Panel();         p2.add( new TextField( "Tex
to", 20) );

        card.add( PanelBoton, p1 );             card.add( PanelTexto, p2);

                add( "Center", card );


                // Para conmutar entre un Card y el otro                ne
w CardLayout().show(card, PanelAwtCard.PanelTexto);             new  CardLa
yout().show(card, PanelAwtCard.PanelBoton);

                }
```

```
                public static void main(String[] args){
                        // No intentar comprender este código por el momento
                        Frame f = new Frame();
                        f.add(new PanelAwtCard());
                        f.show();
                }
}
```

7.5 Actividad independiente

Crear las siguientes interfaces gráficas en un proyecto llamado MisVentanas.

VentanaProductos.java

8 Patrón de diseño Memento. Practica guiada Patrón memento

Memento es un patrón de diseño de **comportamiento que nos permite salvar y restaurar estados anteriores de un objeto** sin revelar datos de la implementación.

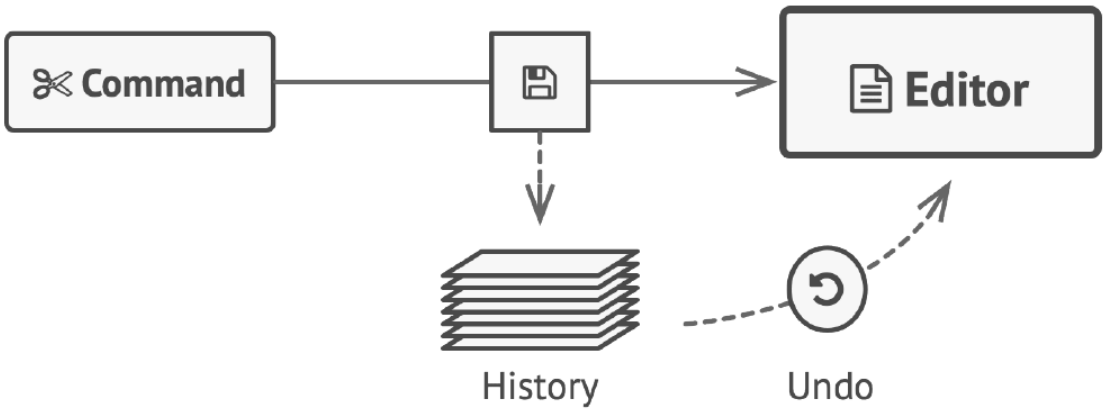
Intent

Sin infringir la encapsulación, captura y externalice el estado interno de un objeto para que el **objeto se pueda devolver a este estado original** o anterior.

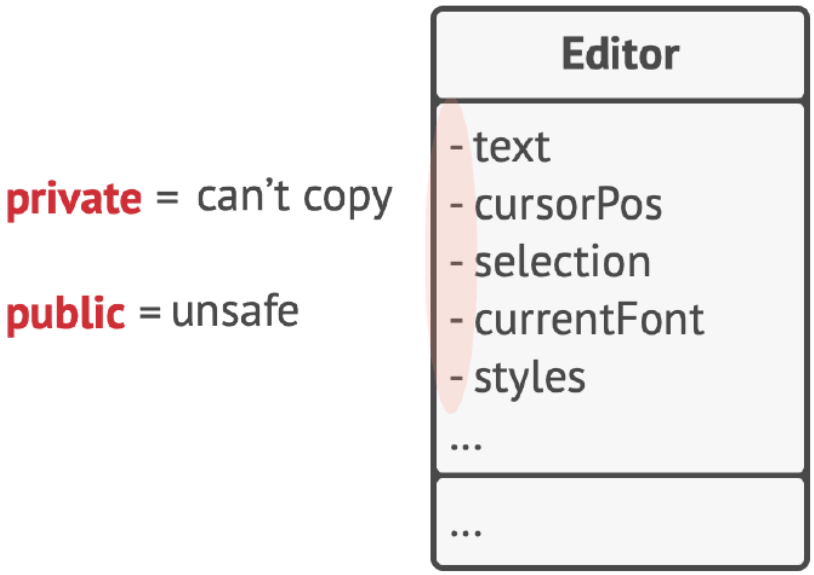
Se comporta **como una magic cookie** (un token o paquete corto de datos pasados entre programas de comunicación) que encapsula una **capacidad de "punto de verificación o restauración".**
Promover **deshacer o rehacer (undo/redo)** al **estado completo del objeto.**

Problema

Imaginad que **estáis creando una aplicación de editor de texto**. Además de edición de **texto simple, su editor puede formatear texto**, insertar en línea imágenes, etc.
En un momento **del desarrollo de la aplicación, se decide permitir a los usuarios deshacer cualquier operación** llevado a cabo en el texto. **Esta característica se ha vuelto tan común** a lo **largo de los años** que **hoy en día la gente espera que cada aplicación tenerlo**. Para el ejemplo teórico de implementación, optamo por un editor. Antes de **realizar cualquier operación, la aplicación registra el estado de todos los objetos y lo guarda en algún almacenamiento**. más tarde **cuando un usuario decide revertir una acción**, la **aplicación obtiene la última instantánea del historial** y la utiliza para **restaurar el estado** de todos los **objetos**.



Pensemos **en esas instantáneas o snapshot del estado**. ¿Cómo se pueden usar? Probablemente **tendríamos que repasar todos los campos de un objeto y copiar sus valores en el almacenamiento**. Sin embargo, **esto sólo funcionaría si el objeto tuviera restricciones de acceso bastante relajadas** a su contenido. Desafortunadamente, la **mayoría de los objetos reales no permiten que otros se asoman dentro de ellos** tan fácilmente, ocultando **todos los datos significativos en campos privados**. Por ahora **vamos a ignorar este problema** y vamos a pensar que nuestros **objetos** de datos o modelo mantienen su estado **con acceso público**. Nos encontraremos el problema de que **nuestro modelo es inseguro**, y nuestro código puede producir errores.



How to make a copy of the object's private state?

Pero hay más. **Analicemos las "instantáneas" reales** del estado del **editor**. ¿Qué datos contiene? Como mínimo, debe contener **el texto real, las coordenadas del cursor, el desplazamiento actual posición,** etc. Para **crear una instantánea, tendría que recopilar estos valores y ponerlos** en algún tipo de **contenedor.**

Lo **más probable es que se vaya a almacenar muchos de estos objetos en contenedores** dentro de **alguna lista que representaría el historial**. Por tanto, los **contenedores** probablemente terminarían siendo **objetos de un clase**.

La **clase no tendría casi ningún método**, pero **un montón de campos que reflejan el estado del editor**. Para permitir que otros **objetos escriban y lean datos hacia y desde una instantánea, probablemente tendremos la necesidad de hacer públicos** sus **campos**. Eso **expondría y haria inseguros los diferentes estados** almacenados del editor, privados o no.
Otras **clases se volverían dependientes en cada pequeño cambio en la clase de instantánea**, que por otro lado **sucede en el estado privado de esta clase** instantanea dentro de campos y métodos privados **sin afectar a las clases externas**.

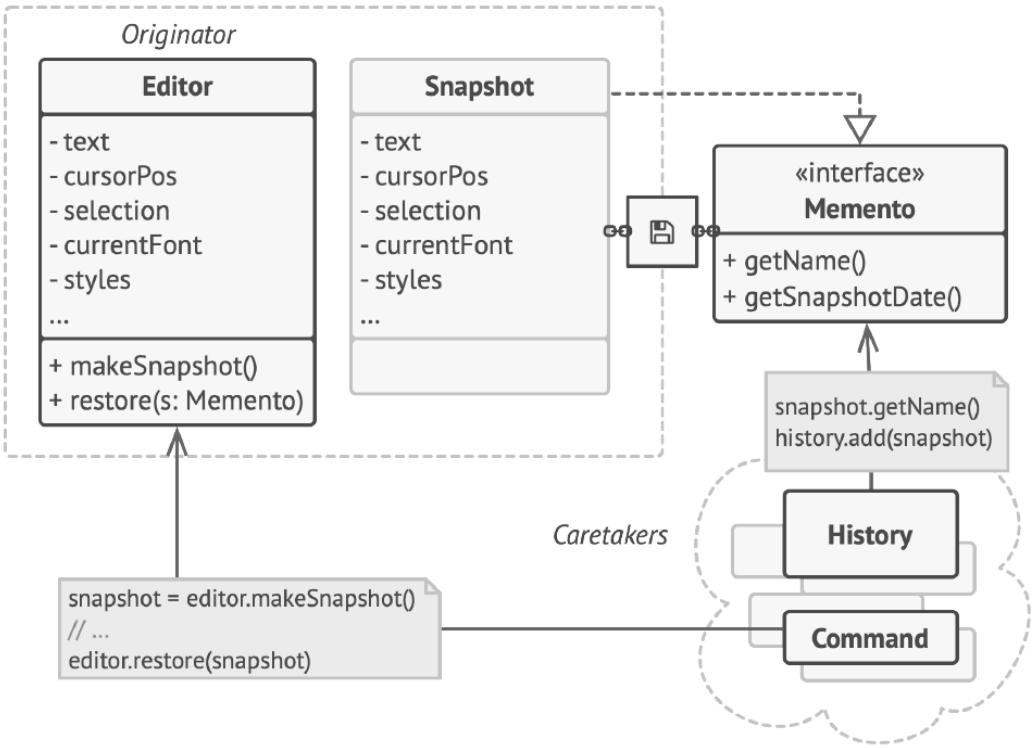
Solución

Todos los **problemas que acabamos de experimentar son causados por saltarnos** el principio de orientación a objetos **encapsulación y revelar estados privados como públicos**. Algunos **objetos tratan de hacer más de lo que se supone** que deben hacer. Para recopilar los **datos necesarios para realizar**

alguna acción, invaden el espacio privado de otros objetos en lugar de dejar estos **objetos realizan la acción real.**

El **patrón Memento delega la creación de las instantáneas de estado al propietario real de ese estado**, el **objeto del que pretendemos guardar su estado**, el **editor**. Por lo tanto, **en lugar de otros objetos que intentan copiar el estado del editor** de el "exterior", **la propia clase de editor puede hacer la instantánea** ya que **tiene pleno acceso a su propio estado.**

El patrón **sugiere almacenar la copia del estado del objeto** en un **objeto especial llamado Memento**. El **contenido del Memento no es accesible desde ningún otro objeto**, excepto el que lo produjo. Otros **objetos deben comunicarse y acceder al objeto Memento utilizando una interfaz limitada** que puede **permitir la obtención de los metadatos** de la instantánea (**tiempo de creación, el nombre de la operación**, etc.), pero no el estado interno completo del objeto original contenido en la instantánea. Hacemos **nuestro código más seguro.**



Esta política tan restrictiva permite **almacenar Mementos dentro de otros objetos, generalmente llamados caretakers**. Ya que el **caretaker trabaja con el memento sólo a través de la interfaz limitada**, **no** es capaz de **manipular el estado almacenado** dentro del memento. Al mismo tiempo, **el originator,** el objeto que origina los estados, nuestro **Editor**, tiene **acceso a todos los campos**

dentro del memento, permitiéndole **restaurar su estado anterior a voluntad.**

En nuestro **ejemplo de editor de texto, podemos crear un historial independiente** de la **clase Editor** para actuar como CareTaker. Una pila de objetos memento almacenados en su interior **que el caretaker permitirá crecer cada vez que el editor** está a **punto de ejecutar una operación**. Incluso **podría renderizar esta pila dentro de la interfaz de usuario de la aplicación**, que **muestra el historial de operaciones** realizadas anteriormente a un usuario.

Cuando **un usuario activa el deshacer, el historial recoge el Objeto memento** más reciente **de la pila y se lo devuelve al editor**, solicitando una deshacer . Dado que el **editor tiene pleno acceso al memento**, cambia su propio estado con los valores tomados del memento. Se podría implementar con una lista doblemente enlazada si nos interesa, poder movernos hacia delante y hacia detrás en el historial.

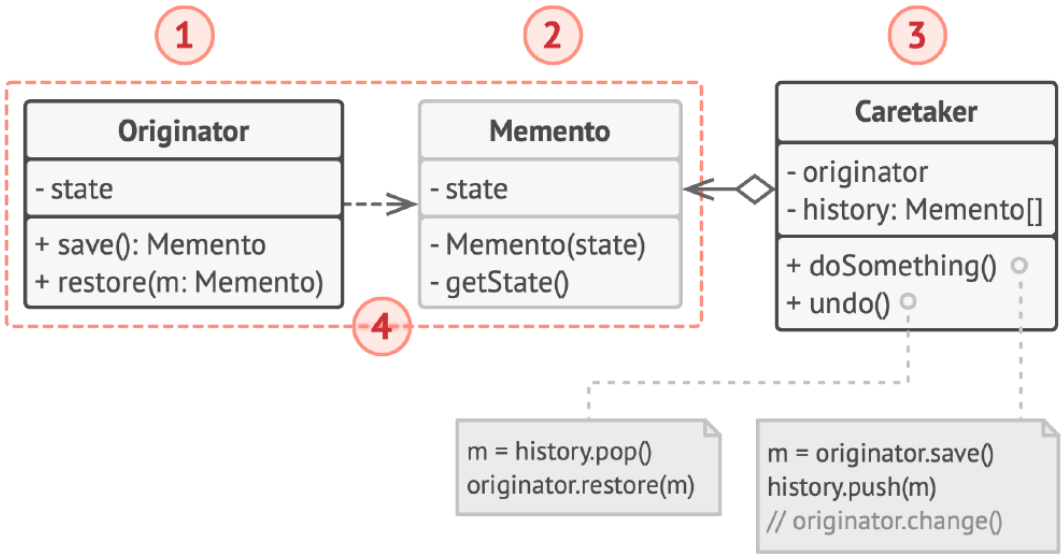
Estructura

Implementación basada en clases anidadas

La **implementación clásica del patrón se basa en apoyarse** en **clases anidadas, disponibles** en muchos lenguajes de programación populares (como C++, C# y Java).

1. La **clase Originator puede producir instantáneas de su propio estado**, así como **restaurar su estado a partir de instantáneas** cuando sea necesario.
2. El **Memento es un objeto de valor que actúa como una instantánea** del **estado del originator**. Es una **práctica común** hacer el Memento inmutable y pasar los datos sólo una vez, a través del constructor.
3. El **CareTaker no sólo sabe "cuándo" y "por qué"** se debe almacenar una **instantanea del estado del originator**, sino también **cuándo se debe restaurar el estado**. Un **CareTaker puede realizar un seguimiento** de la **historia del originator** almacenando **una pila o lista de mementos**. Cuando el originator tiene que **navegar hacia atrás en la historia**, el **Caretaker** recupera el memento más alto en la pila, o el **ultimo elemento de una lista** y se lo **pasa al método que restaura estados** en el objeto **Originator**.

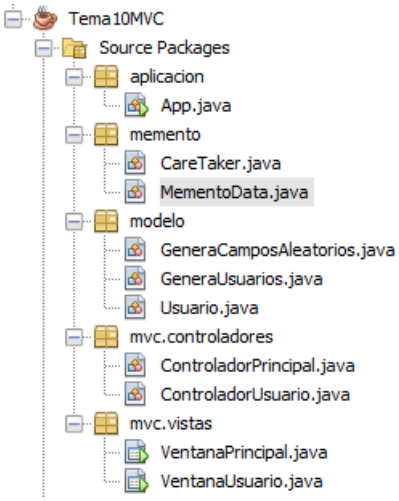
4. En este ejemplo, la clase Originar se anida dentro del objeto de tipo Memento. Realizando **esta acción se permite al Originator acceder a los campos y métodos del Memento**, a pesar de que se declaran privados. En por otro lado, el **CareTaker tiene un acceso muy limitado** a **los campos y métodos del memento**, lo que **le permite almacenar objetos memento en una pila** o **lista**, pero no manipular su estado.



Nota: la **implementación que realizaremos de memento dentro del ejemplo MVC del apartado siguiente es un poco diferente**, **usaremos una lista** pero seguiremos estas pautas.

8.1.1 Ejemplo de Memento y MVC. Primera Parte.

Ejemplo de Memento, dentro **del MVC del apartado siguiente**. Es decir, con un **ejemplo más realista de aplicación vamos a ejemplificar tanto el patrón Memento** como el **MVC**. Para ello **vamos a usar el siguiente proyecto**. Vamos a crear este proyecto completo. Esta vez, **se proporcionará el proyecto con su código** en el aula virtual.



Nuestro Modelo

La **primera clase afectada por añadir el patrón memento** será **Usuario**. Ahora **necesitaremos clonar usuario** para **poder hacer copias del estado de nuestros datos y almacenarlas en un snapshot**. Con **el método clone** podemos **hacer una copia profunda del estado interno de nuestro objeto**, nuestros **modelo de datos**. Pero como veis **el cambio es mínimo** y **no afecta para nada a la implementación o funcionalidad de nuestro modelo**. Nuestra **clase Usuario se seguirá comportando exactamente igual**, es sólo un **método de copia o clonación**.

```
public Usuario clone() {




          return new Usuario(this.id,this.nombre,this.apellidos
,this.Edad,this.horasDeUso,this.numConexiones);

        }
```

Usuario.java

```
package modelo;


import java.io.Serializable;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.TreeSet;
import java.util.function.Function;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
```

```
public class Usuario implements Comparable<Usuario>, Serializable {

	private static final long serialVersionUID = -1234664055657290718L;
	private int id;
	private String nombre;
	private String apellidos;
	private Integer Edad;
	private Double horasDeUso;
	private int numConexiones;

	public Usuario() {



	}

	public Usuario(int id, String nombre, String apellidos, Integer edad,
Double horasDeUso, int numConexiones) {

		this.id=id;
		this.nombre = nombre;
		this.apellidos = apellidos;
		Edad = edad;
		this.horasDeUso = horasDeUso;
		this.numConexiones = numConexiones;
	}

	public int getId() {
		// TODO Auto-generated method stub
		return id;
	}
	public String getNombre() {
		return nombre;
	}
	public String getApellidos() {
		return apellidos;
	}
	public Integer getEdad() {
```

```
                return Edad;
        }
        public Double getHorasDeUso() {
                return horasDeUso;
        }
        public int getNumConexiones() {
                return numConexiones;
        }
        public void setNombre(String nombre) {
                this.nombre = nombre;
        }
        public void setApellidos(String apellidos) {
                this.apellidos = apellidos;
        }
        public void setEdad(Integer edad) {
                Edad = edad;
        }
        public void setHorasDeUso(Double horasDeUso) {
                this.horasDeUso = horasDeUso;
        }
        public void setNumConexiones(int numConexiones) {
                this.numConexiones = numConexiones;
        }



        public String nombreCompleto() {
                return nombre + " " + apellidos;
        }



        @Override
        public String toString() {
                return "Usuario [id=" + id + ", nombre=" + nombre + ", apelli
dos=" + apellidos + ", Edad=" + Edad
```

```
                                    + ", horasDeUso=" + horasDeUso + ", numConex
iones=" + numConexiones + "]";
        }


        @Override
        public int compareTo(Usuario o) {
                // TODO Auto-generated method stub



                return  this.getNumConexiones()>o.getNumConexiones()?1:(this.
getNumConexiones()==o.getNumConexiones()?0:-1);
        }


        public Usuario clone() {



           return new Usuario(this.id,this.nombre,this.apellidos,this.Edad,th
is.horasDeUso,this.numConexiones);
        }




}
```

El objeto Memento y CareTaker

En el **objeto memento vamos a guardar un snapshot o instantanea** de nuestros datos. En este caso **una lista de Usuarios**, que **será mantenida y usada por nuestra clase Originator** que será **la clase ControladorPrincipal del modelo MVC**. Esta **clase dentro modelo MVC está destinada a controlar el acceso a nuestros datos** y el control de nuestras vistas, **nuestras ventanas de aplicación**. Pero **esto lo veremos** más adelante. Ahora **vamos a ver como construimos nuestro objeto Memento y CareTaker**.

Objeto Memento

En **nuestro objeto Memento vamos a guardar nuestro estado actual**, una **lista de usuarios que usará nuestra aplicación** como datos. **ListaUsuario mantiene una copia de nuestros datos**. Además, **guardamos el estado de nuestra snapshot o instantanea**, su **fecha de creación**. Será **suficiente para mantener instantaneas en nuestra aplicación. No necesitamos tantos metadatos** como en **el editor de texto de la teoría**.

```
    List<Usuario> listaUsuarios;
    private Date fechaCreacion;
```

```
public MementoData(List<Usuario> listaUsuario) {
```

Como **podeis ver MementoData es un objeto inmutable**, **no ofrecemos setters para modificar su estado**. Sólo **el constructor para añadir la lista**. Y métodos **getters para recuperar fecha de creación y los datos**, la lista de usuarios.

```
public List<Usuario>  getListaUsuarios()
public Date getFechaCreacion()
```

MementoData.java

```
package memento;


import java.util.Date;
import java.util.List;
import modelo.Usuario;


/**
 *
 * @author carlo
```

```
 */
public class MementoData {
    List<Usuario> listaUsuarios;

    private Date fechaCreacion;

    public MementoData(List<Usuario> listaUsuario) {

        this.listaUsuarios = listaUsuario;

        this.fechaCreacion= new Date();
    }


    public List<Usuario>  getListaUsuarios() {


        return  listaUsuarios;
    }

    public Date getFechaCreacion() {

        return this.fechaCreacion;
    }

}
```

La clase CareTaker

Esta **clase nos va a permitir almacenar un historial de snapshots**, objetos MementoData, para **poder mantener diferentes copias del estado**, y ofrecer **el servicio de undo y redo** a nuestras clases clientes de la aplicación. Siguiendo **las nuevas tendencias de programación** se ha desarrollado todos **los métodos CareTaker como estáticos.**

Inicializamos **el CareTaker con nuestro primer estado Memento**. Guardamos **el snapShot actual en presentSnapshot**, y **lo añadimos a nuestro historial** history este primer snapshot.

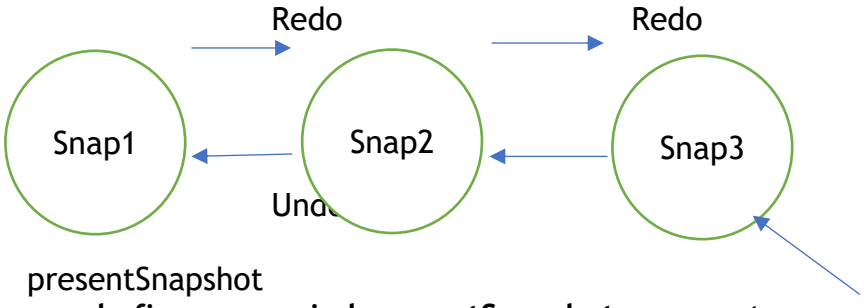
```
        public static void inicializaCareTaker(MementoData datos) {

        presentSnapshot = datos;
        history.add(datos);
```

Usamos una linkedlist para almacenar el historial, pues es **más comoda** para movernos al **inicio y final de la lista.**

```
    private static LinkedList<MementoData>  history = new LinkedList<MementoD
ata>();
```

Tenemos **dos métodos para comprobar si el snapShot actual es el primero o último** de la lista. Si **es el último no podremos hacer Redo**, si **es el primero no podemos hacer Undo.**

```
    public static boolean getEsPrimeroSnapshot() {

        return  history.indexOf(presentSnapshot)==0;

    }

     public static boolean getEsUltimoSnapshot() {

       return   history.indexOf(presentSnapshot)== history.indexOf(history.get
Last());

    }
```



Observar en la figura que si el presentSnapshot que mantenemos **en la clase CareTaker es el último no se pueda hacer Redo**, rehacer. **Si fuera el primero**

de la lista no se podría hacer Undo, deshacer. **Mantenemos el snapshot actual**, los **datos que estamos utilizando en nuestra aplicación** en la propiedad **presentSnapshot**.

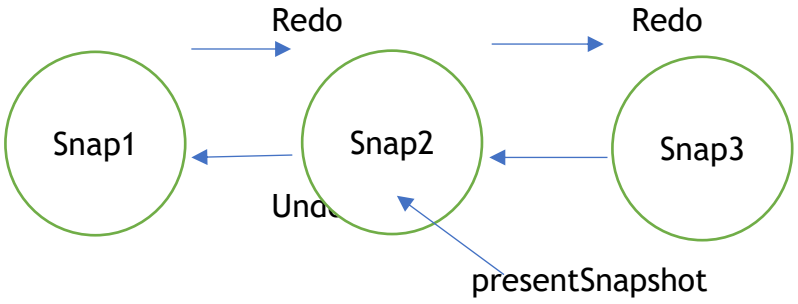
Nota: **Podeis extraer la conclusión de que el patrón Memento es una extensión o tiene muchas similitudes con el patrón State** que ya hemos visto en anteriores temas.

```
private static MementoData presentSnapshot;
```

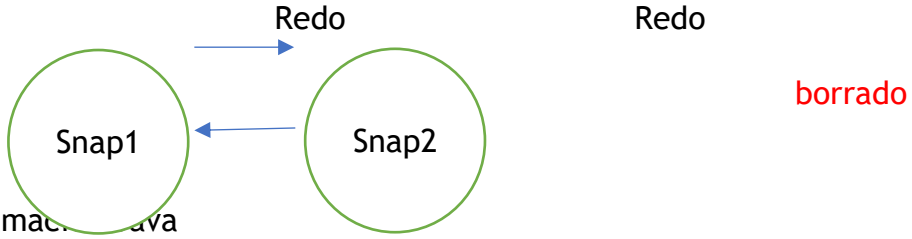
El método **addSnapShot** es llamado **cuando necesitamos realizar un snapShot nuevo**, **cada vez que haya un cambio en nuestros datos**, que llamemos a *guardar datos en nuestra aplicación, creamos un nuevo snapShot*, **clonando el snapShot actual**. Para **clonar la lista, hacemos una copia** `u-> u.clone()` de cada **objeto usuario**, en el Stream. Finalmente, **creamos una lista nueva** con `Collectors.toList()`.

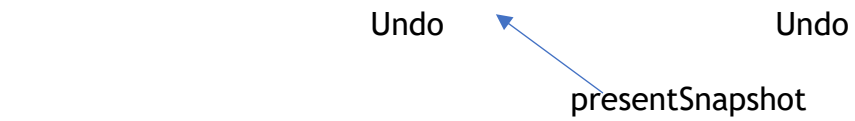
Tenemos que afrontar el problema de realizar un undo, estár en un **estado anterior, y volver a guardar**. Una **solución sería colocar nuestro nuevo estado al final de la lista**. Hemos **optado por otra solución**, **borrar los estados posteriores**, **para practicar nuestro manejo de listas**. Podríamos **quitar estas dos líneas**. Quitaremos estas líneas en la práctica del tema 10.

```
if (!getEsUltimoSnapshot())
    borraHastaFinal();
```

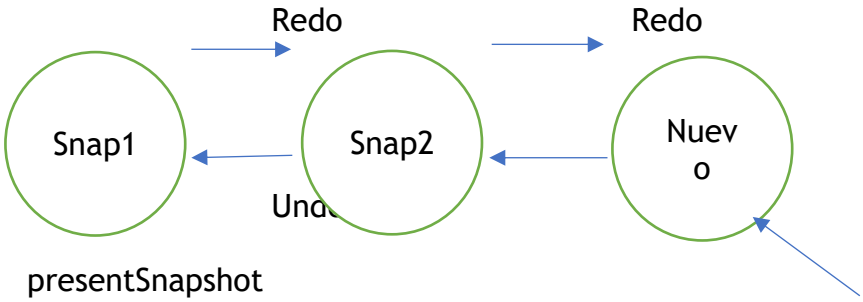


Si **guardamos datos y añadimos uno nuevo a partir de un estado anterior**, borramos **los estados posteriores** al actual.





Y añadimos el nuevo:



Como veis **en el ejemplo un nuevo snapShot** es añadido al final:

```
        history.addLast(memento);
        presentSnapshot=memento;
```

```
    public static MementoData addSnapShot() {



        List<Usuario> listaClonada = presentSnapshot.getListaUsuarios().strea
m().map(u-> u.clone()).collect(Collectors.toList());


        MementoData memento = new MementoData(listaClonada) ;


        if (!getEsUltimoSnapshot())
            borraHastaFinal();


        history.addLast(memento);
        presentSnapshot=memento;
        return memento;
```

```
    }
```

El **método back nos permite ir a atrás en el historial y coger el snapShot inmediatamente anterior** al actual. Si **el snapShot es el primero, devolveríamos el mismo.** Si no es el primero, **devolvemos el snapShot en el historial con índice uno menos que el snapshot actual**, y lo fijamos como **snapShot actual:**

```
            presentSnapshot= history.get(index-1);
```

```
    public static MementoData back() {

        int index =history.lastIndexOf(presentSnapshot);

        if (getEsPrimeroSnapshot())

                presentSnapshot = history.get(index);

        else

            presentSnapshot= history.get(index-1);


        return presentSnapshot;
    }
```

Exactamente tenemos el mismo proceso para el ´metodo `forward()`.

```
public static MementoData forward() {

        int index =history.indexOf(presentSnapshot);

        if (getEsUltimoSnapshot())

                presentSnapshot = history.get(index);

        else
```

```
            presentSnapshot= history.get(index+1);

        return presentSnapshot;
    }
```

El método getSnapshot nos **devuelve el snapShot actual**. Todos estos métodos públicos serán utilizados **por nuestra clase Originator ControladorPrincipal** para **poder realizar undo y redo** en nuestra **interfaz gráfica**.

```
public MementoData getSnapshot() {

        return presentSnapshot;
    }
```

CareTaker.java

```
package memento;

import java.util.LinkedList;
import java.util.List;
import java.util.Stack;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
import modelo.Usuario;

/**
 *
 * @author carlo
 */
public class CareTaker {

    private static LinkedList<MementoData>  history = new LinkedList<MementoD
ata>();
```

```
    private static MementoData presentSnapshot;

    public static void inicializaCareTaker(MementoData datos) {

        presentSnapshot = datos;
        history.add(datos);


    }

    public static MementoData addSnapShot() {


        List<Usuario> listaClonada = presentSnapshot.getListaUsuarios().strea
m().map(u-> u.clone()).collect(Collectors.toList());

        MementoData memento = new MementoData(listaClonada) ;

        if (!getEsUltimoSnapshot())
            borraHastaFinal();

        history.addLast(memento);
        presentSnapshot=memento;
        return memento;



    }




    public static boolean getEsPrimeroSnapshot() {

        return  history.indexOf(presentSnapshot)==0;
```

```
    }

     public static boolean getEsUltimoSnapshot() {

       return   history.indexOf(presentSnapshot)== history.indexOf(history.get
Last());

    }


    public static MementoData back() {

        int index =history.lastIndexOf(presentSnapshot);

        if (getEsPrimeroSnapshot())

               presentSnapshot = history.get(index);

        else

            presentSnapshot= history.get(index-1);


        return presentSnapshot;
    }


    public static MementoData forward() {

        int index =history.indexOf(presentSnapshot);

        if (getEsUltimoSnapshot())

               presentSnapshot = history.get(index);

        else
             presentSnapshot= history.get(index+1);
```

```
        return presentSnapshot;
    }


     private static void borraHastaFinal() {


        IntStream.range(history.indexOf(presentSnapshot)+1,history.lastIndexO
f(history.getLast()) ).forEach(index->history.remove(index));
    }

    public MementoData getSnapshot() {

        return presentSnapshot;
    }

}
```

9 Patrones de diseño en interfaz gráfica

En este tema vamos a explicar básicamente que **es el modelo MVC, Modelo Vista Controlador o View Controller Model** para diseño de aplicaciones con interfaz de usuario **(GUI (graphical user interface))**. Explicaremos **el diseño de GUI** y también incorporaremos **el patrón de diseño Memento**, para implementar **operaciones de Redo y Undo** sobre nuestras aplicaciones.

Como **indicamos anteriormente**, este tema va a **ser bastante práctico**. Su objetivo integrar nuestro modelo de datos y lógica de programa a un **entorno gráfico Swing**, **explicar un modelo de diseño** y un **patrón de diseño** muy **adecuados para GUI**.

10 Modelo Vista Controlador. Practica guiada completa diseño de interfaz con MVC

Modelo-vista-controlador (MVC) es un patrón de arquitectura de software, que **separa los datos y principalmente lo que es la lógica de negocio de una aplicación de su representación** y el **módulo encargado de gestionar los**

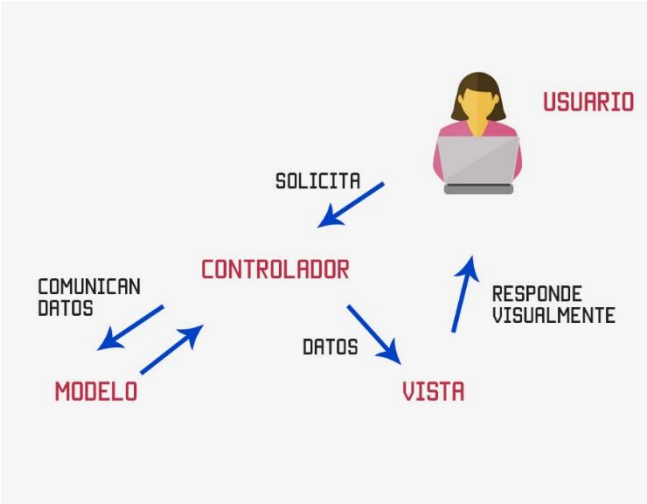
eventos y las **comunicaciones**. Para **ello MVC propone la construcción de tres componentes distintos** que son **el modelo, la vista y el controlador**, es decir, por un lado, **define componentes para la representación de la información**, y por **otro lado para la interacción del usuario**. Este **patrón de arquitectura de software se basa en las ideas de reutilización de código y la separación de conceptos**, características que **buscan facilitar la tarea de desarrollo de aplicaciones** y su posterior mantenimiento.

Como **veréis en el ejemplo el modelo MVC incorpora varios patrones de diseño en su construcción**. Es habitual en los **modelos arquitectónicos de diseño, la combinación de patrones**. Lo **normal es que el MVC incorpore el patrón de diseño Observer**, ya que el **controlador avisa a la Vista (ventana o pantalla) que se registra** en él, acerca de **los cambios**, con el **objetivo de refrescar la vista actual** o bien para cambiar o **navegar hacia una nueva vista**. También **suele incorporar el patrón Strategy, para realizar diferentes algoritmos sobre los datos**, mostrados o procesados.

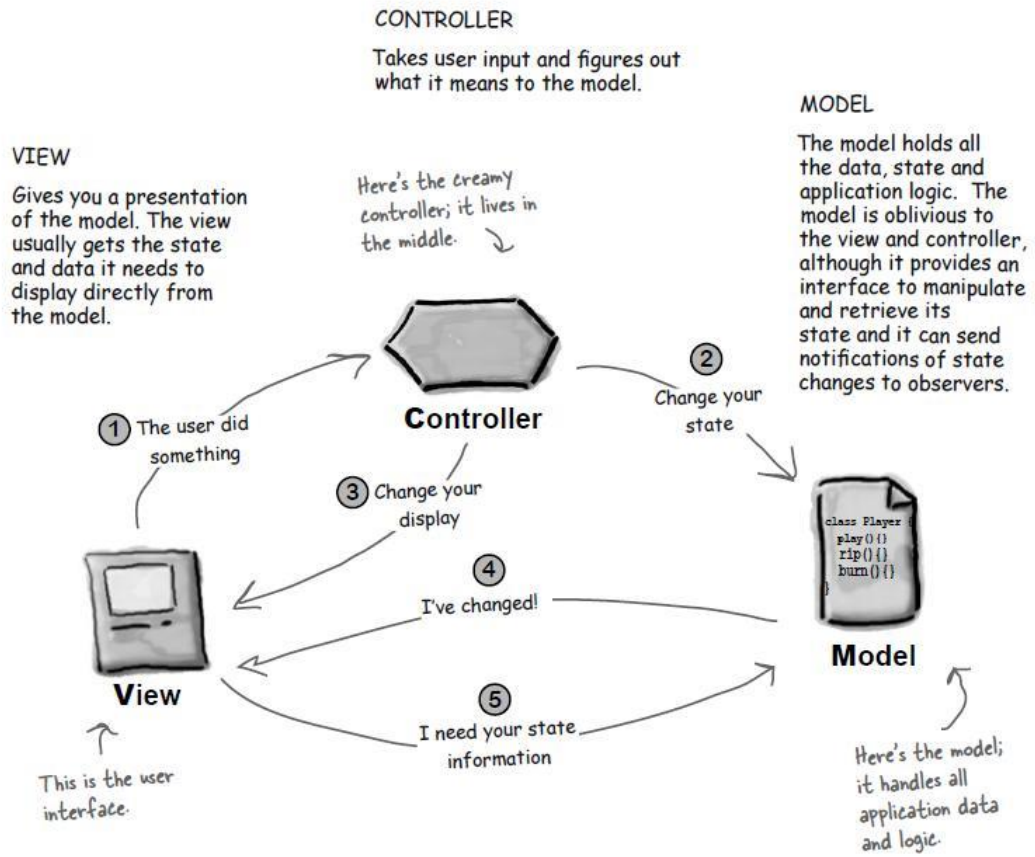
Podéis descomponer el Modelo Vista Controlador como dos patrones vistos ya en unidades anteriores. Observer y Command juntos.

10.1 Componentes

De manera genérica, **los componentes de MVC se podrían definir como sigue:**
El **Modelo**: Es la **representación de la información con la cual el sistema opera**, por tanto, **gestiona todos los accesos a dicha información**, tantas **consultas como actualizaciones**, implementando también **los privilegios de acceso que se hayan descrito en las especificaciones** de la **aplicación** (lógica de negocio). **Envía a la 'vista' aquella parte de la información que en cada momento se le solicita** para que sea mostrada (típicamente a un usuario). Las **peticiones de acceso o manipulación de información llegan al 'modelo' a través del 'controlador'**.
El **Controlador**: **Responde a eventos** (usualmente acciones del usuario) e **invoca peticiones al 'modelo' cuando se hace alguna solicitud** sobre la **información** (por ejemplo, editar un documento o un registro en una base de datos). **También puede enviar comandos a su 'vista'** asociada si **se solicita un cambio en la forma** en **que se presenta el 'modelo'** (por ejemplo, desplazamiento o scroll por un documento o por los diferentes registros de una base de datos), por tanto, **se podría decir que el 'controlador' hace de intermediario entre la 'vista' y el 'modelo'** (**Middleware**).
La Vista: **Presenta el 'modelo'** (**información y *lógica de negocio***) en **un formato adecuado** para interactuar (usualmente la interfaz de usuario), por tanto, **requiere de dicho 'modelo' la información** que debe **representar como salida.**

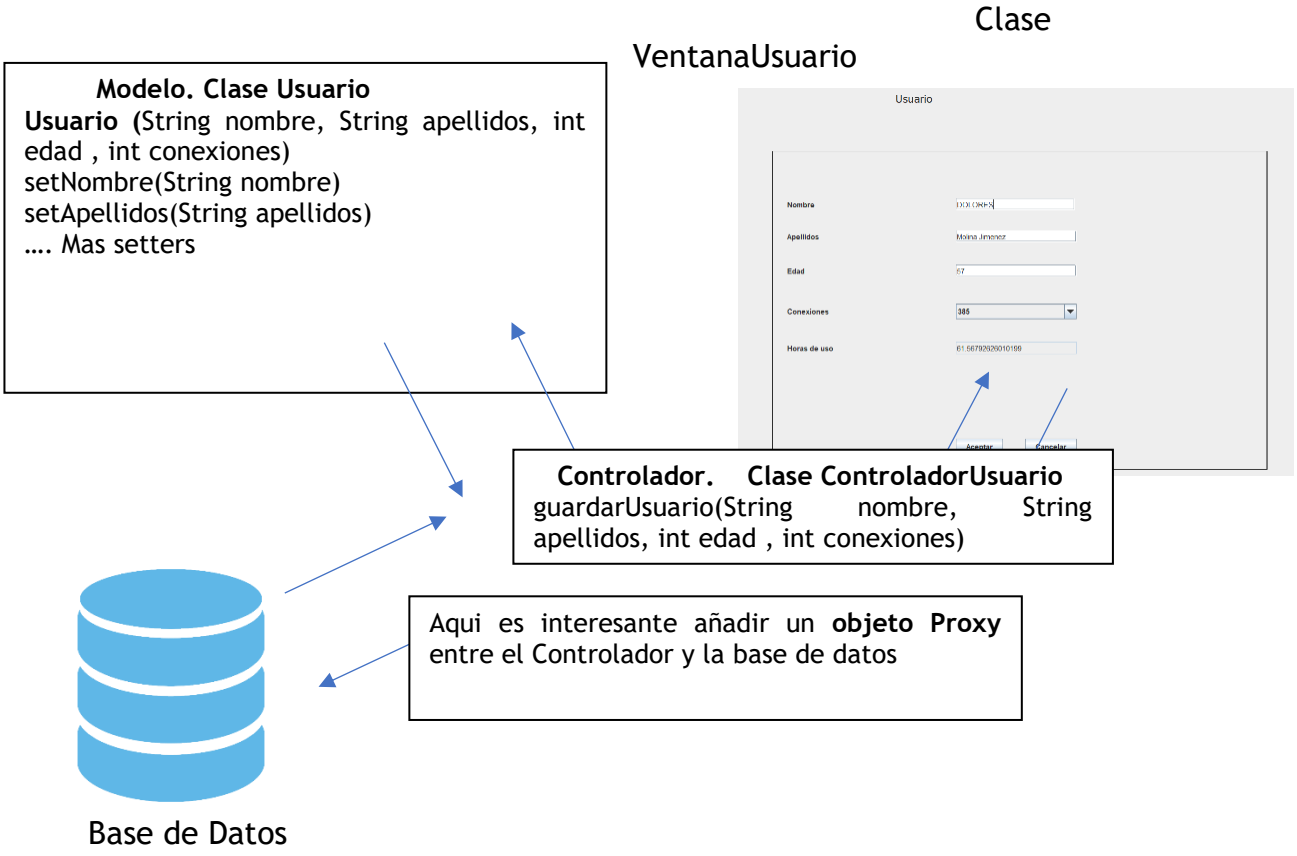


La **siguiente imagen, aunque más gráfica representa un ejemplo claro de las pautas y las acciones de la modelo vista controlador**. El **controlador reacciona a un cambio en los datos o a una acción del usuario** para **cambiar la pantalla**, como veremos en el ejemplo.



El **siguiente gráfico sobre nuestro propio ejemplo también es ilustrativo** de como funcionaría un **MVC en java Swing**.
En nuestro modelo, **la clase Texto, tenemos la información almacenada**. En la **ventana podemos cambiar estos valores**. Pero **la ventana no cambia el**

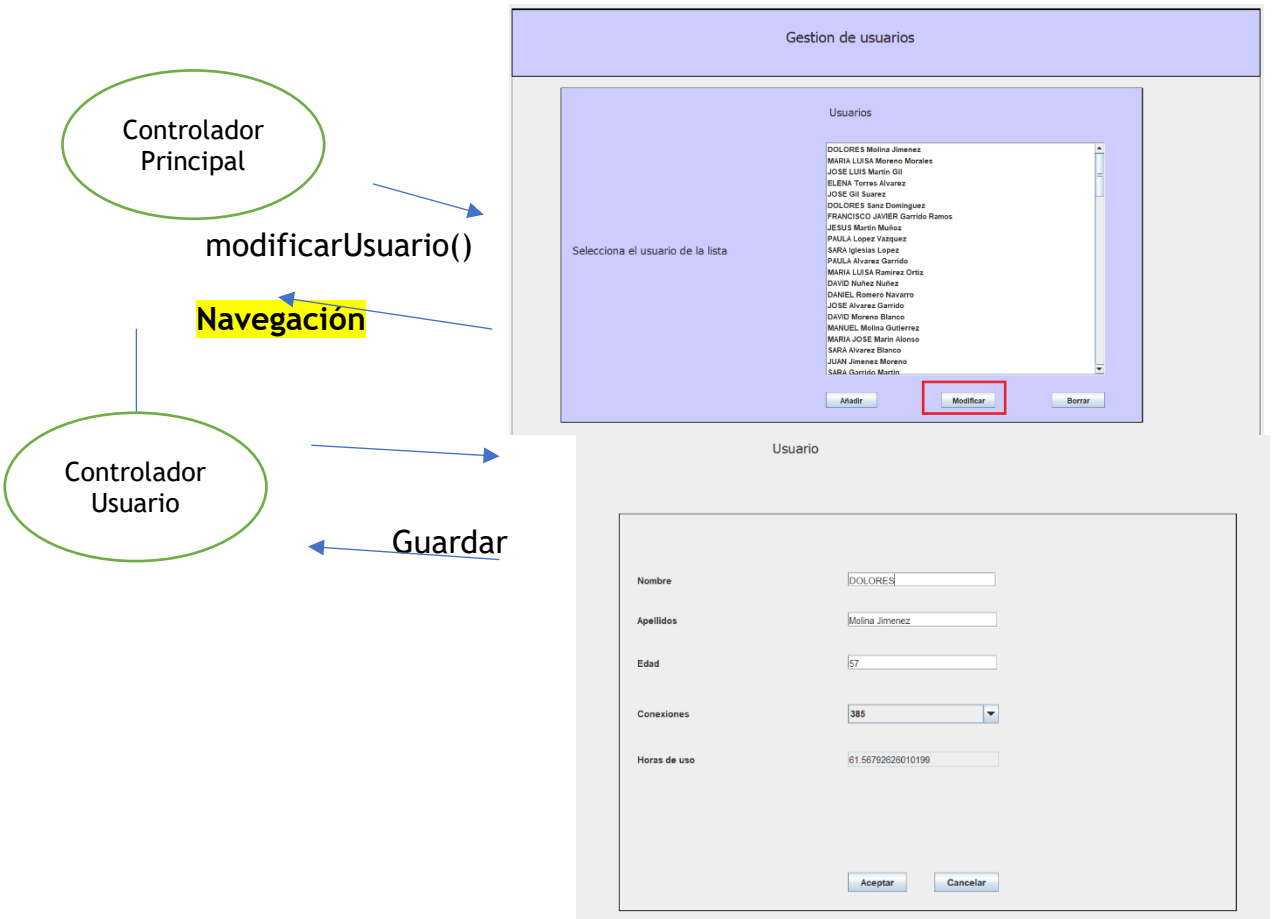
modelo directamente, **es a través del controlador** que **ofrece una serie de métodos** para cambiar nuestro modelo.



Igualmente será el controlador el encargado de interactuar con nuestra base de datos o ficheros, si tuviéramos **que guardar o recuperar la información de base de datos, ficheros o el sistema** que usemos para **almacenar nuestra información**. Como veis es **el elemento de control o central de nuestra aplicación.**

De esta manera **independizamos nuestras vistas**, ventanas, **de nuestros datos**. Un cambio en nuestro sistema de almacenamiento, no afectaría a nuestras Vistas**. Es la clave** del **Modelo Vista Controlador**. Aplicando el patrón Proxy pondríamos otro objeto Proxy entre Base de Datos y Controllador
Igualmente nuestro Controller, en **este caso ControladorPrincipal será el encargado de pasar el control a otros Controllers**. Cuando **pulsamos en el botón modificar de nuestra vista VentanaPrincipal**, **el ControladorPrincipal cede el control a ControladorUsuario**, el **encargado de interactuar con la vista VentanaUsuario**. Es lo que llamamos navegación.
Es **habitual tener un controlador por vista o grupos de vistas**. Cuando **nos referimos a grupos de vistas, nos referimos a por ejemplo un grupo de Vistas que traten con los datos de trabajadores**. Si **tenemos tres vistas que traten sobre los mismos datos, trabajadores**, podríamos **usar un solo controlador**

que controlara las tres vistas.
De manera gráfica **podéis comprobar lo expuesto en los anteriores párrafos en el siguiente gráfico que encontrareis en la página siguiente.**



10.2 Ejemplo completo MVC.

Vamos **a completar la explicación del ejemplo que ya comenzamos con el patrón memento**. Vamos a **explicar primero el funcionamiento completo de la aplicación empezando por la clase App.java.**

La clase App

La **Clase App es el punto de entrada a nuestra aplicación.** Es la clase donde

tenemos **la función Main que se encargará de dar el control a nuestro ControladorPrincipal**, **inicializando el controlador principal**. Es **habitual recuperar los datos de la aplicación en este punto**, o en el **controlador principal,** a voluntad del programador.

```
    public static void main(String[] args) {
        // TODO code application logic here

        ControladorPrincipal.inicializarControladorPrincipal();
    }
```

Clase App.java

```
package aplicacion;

import mvc.controladores.ControladorPrincipal;

/**
 *
 * @author carlo
 */
public class App {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here

        ControladorPrincipal.inicializarControladorPrincipal();
    }

}
```

La clase ControladorPrincipal.

La clase **ControladorPrincipal se va a encargar de la gestión de todos nuestros datos**, la lista de usuarios **dentro del objeto MementoData** que **contiene nuestro SnapShot actual**. Tendremos un **historial de SnapShots de tipo MementoData** en la **clase CareTaker**, como habréis visto anteriormente. **ControladorPrincipal se encargará** de gestionar **estos datos**.

Idealmente **deberíamos añadir un CareTaker** por **cada actividad donde implementemos Undo y Redo**. En el ejemplo **deberíamos añadir un CareTakerUsuario y un MementoUsuario** que contendría un **Objeto Usuario** para la vista **VentanaUsuario**.
Guardamos en **ControladorPrincipal dos propiedades**. Por un lado el **snapShot actual**, que **contendrá la lista de usuarios** con la que **estamos trabajando en la actualidad**. Por otro lado la ventana principal que vamos a iniciar y controlar con este controlador.

En el **método iniciarControladorPrincipal**, cargamos **los datos de nuestra aplicación, que es una lista de usuarios,** generada de **manera aleatoria** y **almacenada en un objeto MementoData**, Iniciamos la ventana principal y la hacemos visible.

```
        public static void inicializarControladorPrincipal() {

         cargarDatos();
         ventana = new VentanaPrincipal();
         ventana.setVisible(true);

    }
```

Como **veis cargarDatos, crea una lista de 100 usuarios**, crea un **Objeto MementoData** snapShot, e inicializa nuestro historial en el CareTaker.

```
       private static void cargarDatos() {



         snapShot = new MementoData( GeneraUsuarios.devueveUsuariosLista(100
));
         CareTaker.inicializaCareTaker(snapShot);
```

```
    }
```

Almacenamos **la ventana principal porque el ControladorPrincipal es un Subscriber**, siguiendo el **patrón Observer**. **Notifica a la ventana principal** cada vez que **hay cambios para que se refresque** y **muestre los cambios**. Por ejemplo, si hacemos un **Undo en la ventana principal, ControladorPrincipal, notifica** a la **ventana principal** de los cambios, y **está los muestra por pantalla**. Si el controlador **se encargará de más de una ventana notificaría** de los **cambios a más de una ventana**. A todas las ventanas que tuviera subscritas.

```
    public static void notificarCambioDatos() {


        ventana.refreshVentana();
    }
```

```
    private static MementoData snapShot;
    private static VentanaPrincipal ventana ;
```

El **controlador se encargá de verificar si se puede hacer un Redo y un Undo** con dos métodos. En la **ventana principal se mostrará como activo Undo y Redo** en el menú Edición sólo **si se pueden realizar**. Tenemos dos métodos que devuelven un booleano, `sePuedeRedo`, `sePuedeUndo`. Se basan en **si el snapshot es o no el primero o el último de la lista** como **vimos anteriormente**.

```
    public static boolean sePuedeRedo () {

        return !CareTaker.getEsUltimoSnapshot();

    }

     public static boolean sePuedeUndo () {

        return !CareTaker.getEsPrimeroSnapshot();

    }
```

Tenemos **dos métodos** redo y undo, para **movernos por el historial** con `CareTaker.forward();` y `CareTaker.back();`y **cargar la versión de nuestros datos** que nos interesa. Además, **notificamos a la ventana principal** de los **cambios** con `controladorPrincipal.notificarCambioDatos();`

```
    public static void redo() {


        snapShot = CareTaker.forward();
         ControladorPrincipal.notificarCambioDatos();


    }
```

```
    public static void undo() {


        snapShot = CareTaker.back();
         ControladorPrincipal.notificarCambioDatos();


    }
```

Obtenemos los datos desde la ventana principal con `getListaUsuarios()` que **devueve la lista de usuarios** contenidas en el **objeto snapShot actual** de tipo **MementoData**.

```
    public static List<Usuario> getListaUsuarios() {

        return snapShot.getListaUsuarios();
    }
```

El **método nuevoSnapShot añade un nuevo snapShot al CareTaker**. Lo vamos a usar **cuando haya cambios en nuestra ventana y tengamos que añadir un nuevo snapShot** al historial.

```
        public static void nuevoSnapShot() {

                snapShot = CareTaker.addSnapShot();

    }
```

Con **el método nuevoUsuario pasaremos el control a ControladorUsuario**, para que **active la ventana VentanaUsuario**, y nos permita **añadir un nuevo Usuario a la aplicación.** Pasando un Optional vacio, **ControladorUsuario sabe que tiene que añadir un usuario nuevo**. Lo **llamamos al pulsar el botón añadir** desde la VentanaPrincipal.

```
    public static void nuevoUsuario() {

        ControladorUsuario.inicializarControladorUsuario(Optional.empty());

    }
```

Con **el método modificarUsuario realizamos la misma labor que en nuevoUsuario**, **pasar el control a ControladorUsuario**. Pero en este caso **pasaremos el Usuario a modificar**, no añadimos un nuevo. A **partir del nombreCompleto buscamos al usuario en la lista de usuarios**. Lo **llamamos al pulsar el botón modificar** desde la **VentanaPrincipal**.

```
public static void modificarUsuario(String nombreCompleto) {



        ControladorUsuario.inicializarControladorUsuario(snapShot.getListaUs
uarios().stream()
                .filter(usuario-> (usuario.getNombre() + " " +usuario.getApe
llidos()).equals(nombreCompleto)).findAny());

    }
```

Borramos el usuario de la lista y refrescamos la VentanaPrincipal notificando

cambios.

```
     public static void borrarUsuario(String nombreCompleto) {


        nuevoSnapShot();



         snapShot.getListaUsuarios().remove(snapShot.getListaUsuarios().strea
m()
                 .filter(usuario-> (usuario.getNombre() + " " +usuario.getApe
llidos()).equals(nombreCompleto)).findAny().get());
         ControladorPrincipal.notificarCambioDatos();


     }
```

Por último **getIdNuevo, devuelve un nuevo Id** para el nuevo usuario creado. **Le suma 1 al máximo de los Id** si **la lista esta llena**, devuelve **uno si la lista esta vacia.**

```
public static int getIdNuevo() {


           if (snapShot.getListaUsuarios().size()==0)
               return 1;
           else
                return
                        snapShot
                         .getListaUsuarios()
                         .stream()
                         .collect(Collectors.maxBy((u1,u2)->u1.getId()>u2.get
Id()?1:(u1.getId()==u2.getId()?0:-1))).get().getId()+1;
       }
```

ControladorPrincipal.java

```
package mvc.controladores;
```

```
import java.util.Comparator;
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;
import memento.CareTaker;
import memento.MementoData;
import modelo.GeneraUsuarios;
import modelo.Usuario;
import mvc.vistas.VentanaPrincipal;

/**
 *
 * @author carlo
 */
public class ControladorPrincipal {

    private static MementoData snapShot;
    private static VentanaPrincipal ventana ;

         public static void inicializarControladorPrincipal() {


          cargarDatos();
          ventana = new VentanaPrincipal();
          ventana.setVisible(true);



    }


    public static boolean sePuedeRedo () {

        return !CareTaker.getEsUltimoSnapshot();

    }
```

```
    public static boolean sePuedeUndo () {
        return !CareTaker.getEsPrimeroSnapshot();
    }

    public static void redo() {

        snapShot = CareTaker.forward();
         ControladorPrincipal.notificarCambioDatos();

    }

    public static void undo() {

        snapShot = CareTaker.back();
         ControladorPrincipal.notificarCambioDatos();

    }
    public static List<Usuario> getListaUsuarios() {
        return snapShot.getListaUsuarios();
    }

        public static void nuevoSnapShot() {

            snapShot = CareTaker.addSnapShot();
```

```
    }


    public static void nuevoUsuario() {

         ControladorUsuario.inicializarControladorUsuario(Optional.empty());

    }

    public static void modificarUsuario(String nombreCompleto) {


        ControladorUsuario.inicializarControladorUsuario(snapShot.getListaUs
uarios().stream()
                .filter(usuario-> (usuario.getNombre() + " " +usuario.getApe
llidos()).equals(nombreCompleto)).findAny());

    }

    public static void borrarUsuario(String nombreCompleto) {

       nuevoSnapShot();


        snapShot.getListaUsuarios().remove(snapShot.getListaUsuarios().strea
m()
                .filter(usuario-> (usuario.getNombre() + " " +usuario.getApe
llidos()).equals(nombreCompleto)).findAny().get());
        ControladorPrincipal.notificarCambioDatos();

    }

      public static int getIdNuevo() {
```

```
            return   snapShot.getListaUsuarios().stream().collect(Collectors.ma
xBy((u1,u2)->u1.getId()>u2.getId()?1:(u1.getId()==u2.getId()?0:-1))).get().ge
tId()+1;
        }



        private static void cargarDatos() {







            snapShot = new MementoData( GeneraUsuarios.devueveUsuariosLista(100
));
            CareTaker.inicializaCareTaker(snapShot);



     }



        public static void notificarCambioDatos() {



            ventana.refreshVentana();
        }

}
```

Reseñable en VentanaPrincipal

No vamos **a colocar el código entero de VentanaPrincipal**, sólo os voy a indicar lo mas **reseñable porque el proyecto se sube entero al aula virtual**. El método

refrescar ventana es muy importante. Se llama **al iniciar la ventana para cargar los datos** y se **llamará cada vez que cambien los datos** para poder **mostrar los nuevos datos** en la ventana. Usamos **la clase DefaulListModel** para **cargar los datos en la lista de tipo JList<String>** el **componente que aparece a la derecha** con el listado de todos los **Usuarios**.

```
            DefaultListModel<String> modelo = new DefaultListModel<String>();
            modelo.addAll(ControladorPrincipal.getListaUsuarios().stream().map(
usuario-> usuario.nombreCompleto()).collect(Collectors.toList()));
            jListUsuarios.setModel(modelo);
```

Si hay un **elemento seleccionado en la lista habilitamos los botones modificar y borrar**. Sino los deshabilitamos.

```
            if (!jListUsuarios.isSelectionEmpty()) {

                btnBorrar.setEnabled(true);
                 btnModificar.setEnabled(true);
else {

                btnBorrar.setEnabled(false);
                btnModificar.setEnabled(false);
```

Si se **puede hacer Redo y Undo habilitamos los submenús Redo y Undo**. Sino los deshabilitamos.

```
            if (ControladorPrincipal.sePuedeRedo())
                jMenuRedo.setEnabled(true);
            else
                 jMenuRedo.setEnabled(false);

            if (ControladorPrincipal.sePuedeUndo())
                jMenuUndo.setEnabled(true);
            else
                 jMenuUndo.setEnabled(false);
```

```
public void refreshVentana() {

          DefaultListModel<String> modelo = new DefaultListModel<String>();
          modelo.addAll(ControladorPrincipal.getListaUsuarios().stream().map(
usuario-> usuario.nombreCompleto()).collect(Collectors.toList()));
          jListUsuarios.setModel(modelo);



          if (!jListUsuarios.isSelectionEmpty()) {

               btnBorrar.setEnabled(true);
                btnModificar.setEnabled(true);

          } else {



               btnBorrar.setEnabled(false);
               btnModificar.setEnabled(false);

          }

          if (ControladorPrincipal.sePuedeRedo())
              jMenuRedo.setEnabled(true);
          else
               jMenuRedo.setEnabled(false);

          if (ControladorPrincipal.sePuedeUndo())
              jMenuUndo.setEnabled(true);
          else
               jMenuUndo.setEnabled(false);

    }
```

En el **evento de seleccion de item en la JList** habilitamos los **botonos de**

modificar y borrar igualmente.

```
  private void jListUsuariosValueChanged(javax.swing.event.ListSelectionEvent
evt) {
        // TODO add your handling code here:


         btnBorrar.setEnabled(true);
          btnModificar.setEnabled(true);
    }
```

Como resumen, **usando el modelo MVC la mayoría del trabajo de lógica de negocio** lo **realizan los Controladores, liberando a la ventana de la mayoría del trabajo**. Es habitual **llevarse los eventos de la ventana también al controlador haciendo que el controlador implemente Eventos como ListSelectionEvent**. Pero **no lo vamos a hacer porque tendríamos** que **dejar de utilizar el Editor de ventanas NetBeans.**

El controlador ControladorUsuario.

Para finalizar **vamos a ver cómo funciona el ControladorUsuario** que es **más sencillo que el ControladorPrincipal**, porque **no hemos hecho que implemente Undo y Redo**, y **además sólo controla un Usuario** en la **VentanaUsuario**.

Lo primero es **la inicialización en el método inicializarControladorUsuario**, guardamos el **Usuario**, **creamos la ventana VentanaUsuario y la hacemos visible**. Como veis **recibe un Optional<Usuario>** como parámetro. Si **el Optional esta vacío es que en VentanaUsuario** añadiremos **un Usuario nuevo**. Si el **Optional está lleno modificaremos el usuario.**

```
         public static void inicializarControladorUsuario(Optional<Usuario> u
ser) {



          usuario=user;
          ventana = new VentanaUsuario();
          ventana.setVisible(true);


    }
```

El **método getDatosUsuario devuelve el Optional** con el **Usuario**.

```
        public static Optional<Usuario> getDatosUsuario() {


            return usuario;
        }
```

El **método más importante es Guardar**. Se **llamará desde la ventana Principal** cuando **guardemos el usuario**. Lo **primero crear un Nuevo snapShot**.

Lo **siguiente si el Optional<Usuario> esta vacío,** creamos un nuevo Usuario y lo **añadimos a la lista de usuarios**.

```
            if (usuario.isEmpty()) {



                usuario= Optional.of(new Usuario(ControladorPrincipal.getIdN
uevo(),nombre,apellidos,edad,0.0, conexiones));


                ControladorPrincipal.getListaUsuarios().add(usuario.get());
```

Si **está lleno, lo buscamos en la lista de usuarios** y lo **modificamos**.

```
else {


            usuario = ControladorPrincipal.getListaUsuarios().stream().filter
(u->u.getId()==usuario.get().getId()).findAny();
            usuario.get().setNombre(nombre);


……
……
```

Para **finalizar notificamos los cambios a la ventana principal** y **volvemos** a la **ventana principal.**

```
ControladorPrincipal.notificarCambioDatos();
```

```
        public static void guardarUsuario(String nombre, String apellidos, i
nt edad , int conexiones) {
            ControladorPrincipal.nuevoSnapShot();



            if (usuario.isEmpty()) {



                usuario= Optional.of(new Usuario(ControladorPrincipal.getIdN
uevo(),nombre,apellidos,edad,0.0, conexiones));

                ControladorPrincipal.getListaUsuarios().add(usuario.get());

            } else {

            usuario = ControladorPrincipal.getListaUsuarios().stream().filter
(u->u.getId()==usuario.get().getId()).findAny();
            usuario.get().setNombre(nombre);
            usuario.get().setApellidos(apellidos);


            usuario.get().setEdad(edad);
            usuario.get().setNumConexiones(conexiones);




             }


ControladorPrincipal.notificarCambioDatos();


        }
```

```
  private static Optional<Usuario> usuario;
    private static VentanaUsuario ventana ;


        public static void inicializarControladorUsuario(Optional<Usuario> u
ser) {
```

```
            usuario=user;
            ventana = new VentanaUsuario();
            ventana.setVisible(true);

    }


        public static Optional<Usuario> getDatosUsuario() {

            return usuario;
        }


        public static void guardarUsuario(String nombre, String apellidos, i
nt edad , int conexiones) {
            ControladorPrincipal.nuevoSnapShot();


            if (usuario.isEmpty()) {


                usuario= Optional.of(new Usuario(ControladorPrincipal.getIdN
uevo(),nombre,apellidos,edad,0.0, conexiones));


                ControladorPrincipal.getListaUsuarios().add(usuario.get());


            } else {


            usuario = ControladorPrincipal.getListaUsuarios().stream().filter
(u->u.getId()==usuario.get().getId()).findAny();
            usuario.get().setNombre(nombre);
            usuario.get().setApellidos(apellidos);


            usuario.get().setEdad(edad);
            usuario.get().setNumConexiones(conexiones);
```

```
            }

            ControladorPrincipal.notificarCambioDatos();

        }
```

Reseñable en VentanaUsuario

Vamos a **ver de esta ventana sólo la carga de datos**, que es lo más importante. Si el **Optional<Usuario> esta lleno, vamos a modificar un Usuario**, por tanto, **rellenamos nuestros campos con los datos de ese usuario**. Hemos supuesto que el **número de Horas no se puede modificar**.

```
private void iniciarDatos() {



        iniciarCombo();
        if (ControladorUsuario.getDatosUsuario().isPresent()) {


            txtNombre.setText(ControladorUsuario.getDatosUsuario().get().getN
ombre());
            txtApellidos.setText(ControladorUsuario.getDatosUsuario().get().g
etApellidos());
            txtEdad.setText(String.valueOf(ControladorUsuario.getDatosUsuario
().get().getEdad()));
            txtHoras.setText(String.valueOf(ControladorUsuario.getDatosUsuari
o().get().getHorasDeUso()));
            jComboConexiones.setSelectedItem(String.valueOf(ControladorUsuari
o.getDatosUsuario().get().getNumConexiones()));



        }
        txtHoras.setEditable(false);
    }
```

En **iniciarCombo JCombox inicializamos la ComboBox** o lista desplegable. Usamos el **Objeto DefaultComBoxModel**, que puede recibir un Array o una colección de tipo Vector. Le pasamos **un Vector** del **numero 0 al 400**, las conexiones mínimas y máximas permitidas. Construimos **el Vector** como veis **usando la API Stream e IntStream**.

```
    private void iniciarCombo() {


        DefaultComboBoxModel<String> modelo = new  DefaultComboBoxModel<Strin
g>(IntStream.range(0, 400)
                                                .mapToObj(i-> String.valueOf(i)).
collect(Collectors.toCollection(Vector<String>::new)));


        jComboConexiones.setModel(modelo);




    }
```

Importante: Falta toda la parte de validación de datos que si haremos en la práctica.
La validación se debe realizar ofreciendo métodos en los controladores para validar el conjunto de datos. El controlador debe a su vez delegar en una clase Validaciones.

11 Ejercicios. Actividades de refuerzo

Ejercicio 1. Añadir las validaciones a la ventana VentanaUsuario.

Vamos a **intentar añadir validaciones a nuestro Ejemplo**.

1. Añadir **un paquete validaciones, y una clase ValidaUsuario**, con **un método estático validaCampos**, que valide el campo Edad, para que sea

un numérico de entre 3 y 140. Para Nombre y Apellidos comprobar que no están vacíos. Devolverá un mensaje de error, tipo “El campo nombre está vacio” o “la edad no es correcta”.

2. Validar **los campos antes de guardar**, en el controlador. Si **los campos de entrada son Incorrectos** que el **ControladorUsuario lance un dialogo de tipo JDialog** indicando que **hay campos incorrectos** y **no se puede guardar el Usuario**.

Ejercicio 2. Añadir a la aplicación el control de errores

Hemos **explicado como usar el Optional y manejar las excepciones** a principio del tema. **Aplicad lo aprendido a la aplicación** anterior:

1. **Cerrando la aplicación si no podemos cargar los datos** completos del sistema, **enviando el Dialogo** adecuado.
2. Recuperandonos de errores menores como **guardar un usuario en el sistema**, o **recuperar un usuario del sistema**, mandando **el dialogo de error adecuado**.

12 Bibliografía y referencias web

Referencias web

Tutoriales de Java Jacob Jenkov

http://tutorials.jenkov.com/

Tutoriales de Java arquitectura Java

https://www.arquitecturajava.com/

Repositorio Github Venkat Subramanian

https://github.com/venkats

Tutoriales Java geeksforgeeks

https://www.geeksforgeeks.org/

Tutoriales Java Baeldung

https://www.baeldung.com/

Bibliografía

Dive Into DESIGN PATTERNS, Alexander Shvets, Refactoring.Guru, 2020

Functional Programming in Java: Harnessing the Power Of Java 8 Lambda Expressions, Venkat Subramanian, The Pragmatic Programmers, 2014

Programación, Alfonso Jímenez Pérez, Francisco Manuel Pérez Montes, Paraninfo, 1ª edición, 2021

Acceso a Datos, Alicia Ramos Martín, Garceta 2ª Edición, 2018

Entornos de Desarrollo, Maria Jesus Ramos Martín, Garceta 2ª Edición, 2020